Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing

Vancouver, British Columbia, Canada
August 10-12 1992

Sponsored by the ACM
Special Interest Group for Automata and Computability Theory and
Special Interest Group for Operating System Principles

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10 AUGUST 1992	FINAL

4. TITLE AND SUBTITLE
PROCEEDINGS OF THE ELEVENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING

5. FUNDING NUMBERS
N00014-92-J-1463

6. AUTHOR(S)
NORMAN C. HUTCHINSON, EDITOR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF COMPUTER SCIENCE
6356 AGRICULTURAL ROAD
VANCOUVER, B.C. CANADA V6T 1Z2

8. PERFORMING ORGANIZATION REPORT NUMBER
ACM ORDER NO. 536920

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
OFFICE OF THE CHIEF OF NAVAL RESEARCH
CODE 1133
BALLSTON TOWER ONE
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

THIS IS THE PROCEEDINGS OF THE ELEVENTH ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, WHICH WAS HELD AUGUST 10-12, 1992 IN VANCOUVER, BRITISH COLUMBIA, CANADA.

14. SUBJECT TERMS
DISTRIBUTED COMPUTING, CONCURRENCY, FAULT-TOLERANT COMPUTING, ALGORITHMS.

15. NUMBER OF PAGES
287

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing

Vancouver, British Columbia, Canada
August 10-12 1992

Sponsored by the ACM
Special Interest Group for Automata and Computability Theory and
Special Interest Group for Operating System Principles

The Association for Computing Machinery
1515 Broadway
New York, New York 10036

Soft Cover ISBN: 0-89791-495-3
Series Hard Cover ISBN: 0-89791-496-1

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

1-800 342-6626
1-410 528-4261 (Outside US, MD and AK)

ACM Order Number: 536929

FOREWORD

The papers in this volume were contributed for presentation at the Eleventh ACM Symposium on Principles of Distributed Computing, held August 10-12, 1992, in Vancouver, British Columbia, Canada. The symposium is co-sponsored by SIGACT and SIGOPS special interest groups within the Association for Computing Machinery. The papers were selected by the program committee from 114 extended abstracts submitted in response to the call for papers. Technical excellence and relevance to the conference were the principal criteria for selection. Submissions were not formally refereed and authors are expected to submit their papers to fully refereed journals.

The program committee would like to thank all the authors who submitted extended abstracts for consideration. We would also like to thank the following colleagues for the assistance in evaluating the submissions: Yehuda Afek, Anonymous, Eran Aharonson, Jonathan Aumann, Baruch Awerbuch, Baruch Awerbuch, Amotz Bar-Noy, Mihir Bellare, Micky Ben Or, Shai Ben-David, Luca Cardelli, Benny Chor Cidon, Rueven Cohen, Shlomi Dolev, Cynthia Dwork, John Ellis, Shimon Even, Limor Fix, Nissim Francez, Matt Franklin, Roy Friedman, Juan Garay, Morrie Gasser, Oded Goldreich, Orna Grumberg, Joe Halpern, Tom Henzinger, Amir Herzberg, Giuseppe Italiano, Kevin Jeffay, Anna Karlin, Shmuel Katz, Charlie Kaufman, Ilan Kesler, Shlomo Kipnis, Hugo Krawzcyk, Shay Kutten, Leslie Lamport, Eliezer Levy, Nati Linial, Tim Mann, Alain Mayer, Dah Ming Chiu, Shlomo Moran, Yoram Moses, Moni Naor, Seffi Naor, Noam Nisan, Yoram Ofek, Rafail Ostrovsky, Krisha Palem, Boaz Patt-Shamir, David Peleg, Steven Ponzio, Ophir Rachman, K.K. Ramakrishnan, Tom Rodeheffer, Ronny Roth, Vijay Saraswat, Baruch Schieber, Hadas Shachnai, Nir Shavit, Yuval Shavitt, Taly Shintel, Bob Simcoe, Ray Strong, Bob Thomas, Mark Tuttle, Moshe Vardi, George Varghese, Orli Waarts, Jennifer Welch, and Shmuel Zaks.

PROGRAM COMMITTEE

ORGANIZING COMMITTEE

CONFERENCE CHAIR

Norman C. Hutchinson
University of British Columbia

PROGRAM CHAIR

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Lab

LOCAL ARRANGEMENTS AND REGISTRATION CHAIR

Norman C. Hutchinson
University of British Columbia

TREASURER

Richard Anderson
University of Washington

SPONSORS

Office of Naval Research

Centre for Integrated Computer Systems Research
NSERC
IBM Almaden Research Center

Table of Contents

Session 1:

Session 2:

Session 3:

Session 6:

Session 7:

Distributed Priority Algorithms Under One-Bit-Delay Constraint

Reuven Cohen*
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA

Adrian Segall
Dept. of Computer Science
Technion, Haifa 32000, Israel

Abstract

The paper deals with the issue of station delay in token-ring networks. It explains why one-bit-delay is the minimum possible delay at every station and shows that the station delay depends on the distributed computations performed in the ring. Then, the paper introduces the distributed priority mechanism for token-rings, as approved by the IEEE-802.5 standard. This mechanism attaches to the token, that circulates around the ring and controls the access to the shared medium, a priority field P and a reservation field R. These two fields work together in an attempt to match the service priority of the ring to the highest priority message that is waiting to be sent.

It is shown that due to the computation restrictions imposed by the one-bit-delay requirements, this priority mechanism has a grave deficiency as follows. When the token priority is higher than the maximum reservation ($P > R$), the token should make up to $\mathcal{P}$ round-trips, where $\mathcal{P}$ is the number of priority levels, before P is reduced to R. During this time period, no station may seize the token and send a message. This leads to loss of bandwidth.

The paper presents a new priority mechanism that retains the desired properties of the standard. However, in the new protocol when $P > R$ holds, P is reduced to R in at most 1 round-trip rather than in up to $\mathcal{P}$ round-trips.

1 One-Bit-Delay In Ring Networks

In a token-ring[1] Local Area Network (LAN), the stations are located on a directed ring and each station transmits to its downstream neighbor. A short control message, called *token*, circulates around the ring and regulates the access to the shared medium.

Each station in the ring can be in REPEAT or TRANSMIT mode. A station in REPEAT mode transmits to its downstream neighbor almost the same bit stream it receives from upstream. However, every received message contains a header with several bits that can be changed by stations in REPEAT mode. These bits will be called *control bits*.

A station in REPEAT mode that has a message to send waits to receive the token. When the token

*This work was conducted in part when this author was with the Dept. of Computer Science, Technion, IIT.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0001...$1.50

[1] In this paper we consider the 'traditional' token-ring networks, that use moderate transmission rate — up to 16 Mb/s. In high-speed token-rings (like the 100 Mb/s FDDI), one-bit-delay is neither possible nor necessary.

s received, the station does not transmit it to its lownstream neighbor. By this action, the station eizes the token and gets transmission rights, so it :an send its waiting message into the ring. The nessage circulates around the ring and is copied)y its destination (that does not remove it). When he sender receives the message back after one ring evolution, it removes the message from the ring, ssues a new token and returns to REPEAT mode. Consequently, the next station to have the opportuiity of gaining transmission rights will be its downstream neighbor.

This access control scheme implies that at any given time at most one station can be in TRANSMIT mode. At the time when some station is in TRANSMIT mode, all the other stations are in REPEAT mode, repeating the message sent by the former. This ensures that every sent message will be received by its destination, complete a round-trip and return to its sender.

Consider a station in REPEAT mode. Let B_1^- be the first bit the station *receives* after entering this mode, B_2^- be the second incoming bit and so forth. Similarly, let B_1^+ be the first bit the station *transmits* after entering REPEAT mode, B_2^+ be the second bit and so forth. Recall that for every i, if B_i is *not* a control bit, then B_i is repeated (i.e, $B_i^+ \leftarrow B_i^-$) and $B_i^+ = B_i^-$ holds. However, if B_i is a *control bit* then B_i^+ can be different from B_i^-.

The station delay is defined as the interval between the time the station starts receiving a bit and the time it starts transmitting the same bit, either with or with no change. It is convenient to express the station delay in terms of one bit time; namely, the time required to transmit one bit ($1/T$, where T is the transmission rate). A station that starts transmitting B_i^+ at the time when it starts receiving B_i^- introduces no delay. On the other hand, a station that starts transmitting B_i^+ at the time when it starts receiving B_{i+j}^- works with delay of j bits.

When the transmission rate in the ring does not exceed several Megabit/s, the delay introduced by the stations is the main factor of the round-trip delay. For instance, consider a 4 Mb/s token-ring LAN with 100 stations. Suppose that the average distance between every two neighbors is 10m. Assuming propagation delay of $2 \cdot 10^8 m/sec$, such a ring would have a round-trip delay of

$$(100 \cdot 10)\frac{4 \cdot 10^6}{2 \cdot 10^8} + 100 \cdot D = 20 + 100 \cdot D \text{ bits}$$

where D is the delay at each station. This means that the propagation delay is only 20 bits, and that the delay in a ring whose stations have j-bit delay is almost j times the delay of a ring whose stations work with one-bit-delay.

Minimizing the round-trip delay is an important object in token-rings, since in such networks the throughput increases as the delay decreases [1]. Moreover, a token-ring whose round-trip delay is small can support the transmission of real-time data.

A token-ring station cannot work with zero delay. Zero delay can be achieved only in a 'dead' station, which is not expected to change the control bits. Such a station is short-circuited and, therefore, it is bypassed by the signal. An operational station, however, is never short-circuited since it is expected to change some of the received bits. Therefore, in order to repeat a bit, an operational (i.e, not short-circuited) station must completely receive the bit and then transmit it. One may consider a '0-delay scheme' according which a station is short-circuited before receiving a bit that should be repeated and is reconnected before receiving a bit that should be changed. However, this would

not work since switching to and from short-circuit mode takes time and results in loss of many bits.

The conclusion is that the delay at every operational station is of at least one bit. However, in order to work with one-bit-delay, a station must be able to start transmitting every *control bit,* whose outgoing value is not necessarily equal to its incoming value, as soon as it completes receiving the corresponding bit. *This implies that when some control bit is not repeated, its outgoing value must be determined independently of its incoming value and of the value of subsequent incoming bits.*

For example, suppose that B_i is a control bit. Suppose also that the station starts receiving this bit at t, thus completes receiving it at $t+1$. If the value of B_i^+ is a function of the value of B_i^-, the station cannot start computing this function before $t+1$, when B_i^- is known. Therefore, the station cannot start transmitting B_i^+ at $t+1$, but only after the computation is completed. If the outgoing value of a bit depends on the incoming value of subsequent bits, the station delay increases accordingly.

Conclusion: When the ring stations work with one-bit-delay, every outgoing bit B_i^+ is either a repetition of the corresponding incoming bit (i.e. $B_i^+ \leftarrow B_i^-$) or can be expressed as a function $\mathcal{F}$ that is not dependent on the values of B_i^-, $B_{i+1}^-, \cdots$

Note that the decision of a station that works with one-bit-delay as to whether to repeat B_i, or to compute and transmit this bit according to some function $\mathcal{F}$ that is not dependent on B_i^-, $B_{i+1}^-, \cdots$, can be made according to the value of B_{i-1}^-, $B_{i-2}^-, \cdots$. For instance, consider the problem of calculating and transmitting the maximum of an incoming N-bit string, that represents a binary number, and a local one. Let $\{B_1^-, B_2^-, \ldots, B_N^-\}$ be the incoming string, where the first bit is the most significant one, and $C = \{C_1, C_2, \ldots, C_N\}$ be the stored number. For example, for $N=3$, if the input string is 011 and C is 100, the outgoing string should be max(011, 100) = 100, whereas if $C = 001$, the outgoing string should be 011. The following one-bit-delay algorithm calculates and transmits the maximum:

$i \leftarrow 1$

$\star$ if $C_i = 1$ then $B_i^+ \leftarrow 1$ else $B_i^+ \leftarrow B_i^-$

if $B_i^+ \neq B_i^-$ then $B_{i+1}^+, \ldots, B_N^+ \leftarrow C_{i+1}, \ldots, C_N$

else if $B_i^+ \neq C_i$ then $B_{i+1}^+, \ldots, B_N^+ \leftarrow B_{i+1}^-, \ldots, B_N^-$

else $i \leftarrow i+1$; if $i < N$ go to $\star$

The minimum of an incoming string and and a local one can be computed and transmitted similarly.

As explained so far, a lower bound on the station delay is imposed by the rules that determine the outgoing values of the *control bits*. Since these rules are part of the Medium Access Control (MAC) protocol, it can be said that the station delay depends on the access control protocol executed in the ring.

In the simplest version of the token-ring access control protocol, all messages have the same priority. This implies that a station with a waiting message can seize the token upon receiving it and can transmit its message into the ring under no further restriction. Since the token is released by the same station after the latter completes a ring revolution, this simple mechanism ensures fairness in the sense that the transmission rights are passed from each station to its downstream neighbor. References [2] and [5] deal with one-bit-delay implementation of this protocol. However, in order to support multiple services with different time requirements, like real-time voice samples, interactive data, files transfer and so forth, most Local

Area Networks use some means of priority mechanism [6].

The rest of the paper deals with the design of distributed priority mechanisms in token-rings under the restriction of *one-bit-delay*. Section 2 presents the priority mechanism for token-rings as approved by IEEE-802.5 standard. It shows that this mechanism was designed such that the ring stations would be able to work with *one-bit-delay*. However, due to the computation restrictions imposed by the one-bit-delay requirements, this priority mechanism may result in loss of bandwidth and starvation.

Section 3 presents an alternative distributed scheme. The new mechanism retains the fairness and liveness properties but it eliminates the deficiencies associated with the standard protocol. The new protocol has been designed for one-bit-delay operation as well. However, it uses several properties of the protocol variables to ensure that the outgoing value of some bit B_i is the correct function of the received value of subsequent bit B_j, where $j > i$, *without waiting to receive* B_j. The formal specification of the new protocol is presented in Section 4 and its main properties are summarized in Section 5. Section 6 concludes the paper.

2 Priority Mechanism in Token-Rings

The distributed priority mechanism for token-rings, as approved by IEEE-802.5 standard, can be summarized as follows [4].

1) The *token* (also called *token-frame*) consists of three fields,[2] as shown in Figure 1(a): the Priority field P (3 bits), the Token bit T which is always 0, and the Reservation field R. A station that seizes the token changes it into a *data-frame* by setting the token bit T to 1 and appending Destination, Source and Data fields. The first two fields contain the identity of the destination and the sender, respectively. The Data field contains the message to be sent. Figure 1(b) shows the structure of a data-frame.

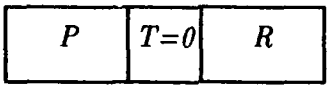
2) Each message is associated with a priority. The most urgent messages have priority 7, whereas the least urgent messages have priority 0. The priority range can be extended by increasing the appropriate fields (P and R).

3) A station wishing to send a message with priority Pm must wait for a token ($T = 0$) with $P \leq Pm$. When such a token is received, the station converts it into a data-frame (by setting $T \leftarrow 1$ and appending Source, Destination and Data fields) and resets the reservation field R to 0. In addition, the station changes its mode from REPEAT to TRANSMIT.

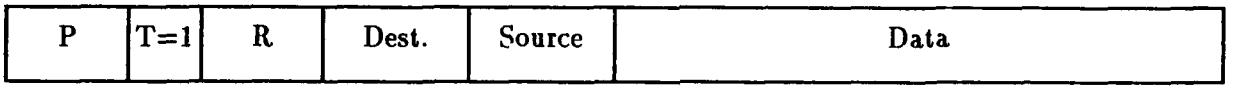
4) While waiting for a usable token (i.e, a frame for which $T^- = 0$ and $P \leq Pm$ holds), a station may reserve a future token with the required priority Pm by setting the R field to Pm in every token- and data-frame it receives, provided that the received R is not larger than Pm. This means that such a station should perform $R^+ \leftarrow \max(R^-, Pm)$. As shown before, this can be done with one-bit-delay.

5) A station in TRANSMIT mode waits to receive the frame with its message back, after the latter completes a ring revolution. When the data-frame is received back, the station issues a new token by changing T back to 0 and removing the Destination, Source and Data fields. The priority field P of the new token is set to the maximum of R^- (the reservation field in the received data-

[2] In fact there is an additional one-bit field, called M, which is required for recovery. However, it plays no role in the priority mechanism, and therefore it is disregarded. Some irrelevant fields in the data-frame, as the CRC, are omitted as well.



(a) Token-Frame



(b) Data-Frame

Figure 1: a Token- and a Data-Frame in IEEE-802.5 Token-Ring

frame), P^- (the priority field in the received data-frame) and Pm (the priority of the most urgent message the station needs to send). This implies that when a data-frame is changed into a token-frame, $P^+ \geq P^-$ holds.

6) A station that increases the service priority of the ring, while changing a data-frame containing its own message to a token-frame, is responsible for returning the priority to its former level later. This prevents *livelock* situations, where the token circulates the ring indefinitely just because its priority level P is higher than the priority of any waiting message. As explained later, this is also the key means for ensuring *fairness* among stations.

In order to be able to restore the old value of P, a station that increases P stores the old and new values in local variables Sr and Sx, respectively. Later, when the station detects a token with priority $P = Sx$, it decreases the priority, unless it has a waiting message with $Pm \geq P$. The new value P^+ is the maximum of the priority of the token before the station had increased it (as stored in Sr), the maximum reservation R^-, and the priority of the most urgent local message Pm. If a station increases the priority from p to p' say, and later decreases it to an intermediate value p'', it replaces the value p' in Sx with p''. This will enable the station to later decrease the priority from p'' to p or again to an intermediate value.

Since a station may increase the priority level more than once before decreasing it, Sx and Sr should be *stacks*. Values are pushed into the stacks every time the station increases P and are popped when the priority is decreased to the *old* level. When a station decreases the priority level to an intermediate value, the latter replaces the top of Sx, while Sr is unchanged.

This approach, where only a station that had increased the priority from p' to p'' can decrease it from p'' to p', in one or more steps, is the basic means for achieving *fairness*. It ensures that if the priority increases and then drops to its previous value p, *the first station to benefit from the new level is the downstream neighbor of the station that has increased* P.

A station that increases the token priority is called a *stacking station*. It remains in this category as long as its stacks are not empty.

Almost all the above operations can be implemented with one-bit-delay. The only difficulty is at stacking stations. Recall that whenever it receives a frame, such a station is expected to do the

following operation:

"if $(T^- = 0) \wedge (P^- = Sx) \wedge (P^- > Pm)$

then $P^+ \leftarrow \max(R^-, Sx, Pm)$"

Since T and R come after P, this operation *cannot* be done with one-bit-delay. The IEEE-802.5 standard copes with this difficulty in the following way. A stacking station that receives a token- or a data-frame repeats P with no change ($P^+ \leftarrow P^-$). After knowing P^-, the station checks whether $Pm \geq P^-$ holds, in which case the station sends the message provided that $T^- = 0$.

However, if $Pm < P^-$, the station compares P^- with the top of its Sx stack. If the two values are equal, the station must decrease P, provided that $T^- = 0$. Therefore, immediately when $T-$ is completely received, the station transmits $T^+ = 1$ and then tests the received value of T. As explained in Section 1, this is a one-bit-delay operation although T is not repeated, because T^+ is determined independently of T^- and subsequent bits.

If T^- happens to be 1, namely the station is not supposed to alter P, it transmits the rest of the incoming frame with no change. In such a case the station works with one-bit-delay and performs exactly what it is expected to: repeat P and T with no change.

On the other hand, if the value of T^- is 0 and $P^- = Sx$, the station has done 2 'mistakes': it has transmitted P with no change, rather than setting it to the maximum of R^- and Pm and it has changed T to 1 rather than repeating it with no change. These two mistakes are corrected in the following way. The station generates a *new* token-frame with the appropriate priority field. Then it waits to receive the first 'frame', which is neither a token-frame nor a data-frame, and removes it from the ring.

The above procedure enables a stacking station to operate with one-bit-delay as long as it should not change P. Namely, when (1) the station receives a token, but the top value of the Sx stack and the priority of the token are different ($Sx \neq P^-$) or (2) the station receives a token, but it has a waiting message with a priority equal to or greater than the priority of the token ($Pm \geq P^-$) or (3) the station receives a data-frame ($T^- = 1$).

On the other hand, when the stacking station should decrease the priority, it not only introduces higher delay, but also temporarily generates a second frame. As explained in [2], in many cases frame multiplication, even temporary, is unacceptable or at least undesirable.

However, the main deficiency of the standard protocol is that *when $P > R$ the token may make up to $(P - R)$ round-trips before P is reduced to R.* Consider a token with priority 7 and suppose that the value in the reservation field is 0, indicating that P should drop to 0. Suppose also that there are seven stacking stations, each of which have increased the token by one level. In such a case, the priority will be decreased in 7 steps as follows. In the first step, the stacking station that has upgraded the priority from 6 to 7 decreases it to 6. In the second step, the priority is decreased from 6 to 5, and so forth. The number of round-trips required to achieve $P = 0$ depends on the relative location of the stacking stations. In the worst case (where the station that decreases P from 7 to 6 is the downstream neighbor of the station that decreases P from 6 to 5, and the latter is the downstream neighbor of the station that decreases P from 5 to 4, and so forth), almost 7 round-trips are required.

In those cases where several round-trips are required to reduce P to R, no station may seize the token and send messages since P is too high.

Therefore, the ring bandwidth during this time interval is lost. Moreover, while the priority crawls to the desired level, new urgent messages may become ready for transmission. These messages may cause the priority field to increase before the least urgent messages are served. Since such a condition may repeat, the messages with lower priority might have to wait forever.

It should be clear that the only reason for the gradual decrease of P to R is enabling the stacking stations to operate most of the time with one-bit-delay as shown above. If this restriction were eliminated, namely stacking stations were allowed to receive and interpret the incoming value of T and R before determining the outgoing value of P, it would have been possible to reduce P to R in at most 1 round-trip while preserving the fairness and liveness properties.

3 The New Mechanism

The new mechanism is modification of the IEEE-802.5 one. While it can be performed with one-bit-delay and it ensures fairness and liveness in the same sense the IEEE-802.5 protocol does, the new protocol *reaches the required priority level considerably faster than in the standard, without unnecessarily going through intermediate levels.* The only penalty in the new protocol is an extra 3-bit field in the token and data-frames, which is insignificant.

The basic elements of the new protocol are as follows. **(1)** The token and data-frames contain a token bit T, a priority field P and *two* reservation fields, R_1 and R_2, as shown in Figure 2. **(2)** Each station manages a local vector of 8 entries, instead of two stacks. This vector plays the same role as the stacks in the original mechanism, namely ensures fairness and liveness. **(3)** The rules for increasing the priority levels are the same as in the standard. The decreasing rules, however, are different.

In the new protocol, the maximal reservation is held in the first reservation field R_1 of a token ($T = 0$), and in the second reservation field R_2 of a data-frame ($T = 1$). The second reservation field R_2 in a token-frame, and the first reservation field R_1 in a data-frame play no role. When a station that wishes to send a message with priority Pm receives a frame (either a token- or a data-frame), it increases R_1 to Pm, provided that $R_1^- < Pm$. Since when the station receives R_1 it does not know yet the value of the token bit T, the reservation in R_1 is only tentative. If the station recognizes later that $T^- = 1$, it makes another reservation in R_2.

After trying to increase R_1, a station that wishes to send a message receives the priority field P and tests whether its value is less than or equal to the priority of its waiting message Pm. If $Pm \geq P^-$ and the station does not change P as explained later (i.e, $P^- = P^+$) and $T^- = 0$, the station is allowed to send its message. Suppose that $Pm \geq P^-$ and $P^- = P^+$ hold. At the time when T^- is completely received, the station starts transmitting $T^+ \leftarrow 1$ independently of T^- (this is a one-bit-delay operation), and tests the received value of T. If $T^- = 1$, the station cannot send its message since the received frame is not a token. Thus, it only tries to make a reservation in R_2. However, if the received value of T is 0, the station resets R_2 to 0 and sends its message. Since R_2 is reset to 0 whenever a token-frame is converted into a data-frame (like R in the original protocol), when the latter is received back by the sender, it contains the maximal reservation made by the stations in the last round-trip. Note that R_1 cannot be used to indicate the required reservation, since from one-bit-delay considerations it cannot be re-

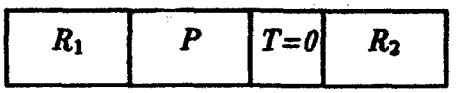


Figure 2: a Token-Frame in The New Scheme
(for a Data-Frame, $T = 1$ holds and Destination, Source and Data field are appended)

set to 0 when the station knows that the received frame is a usable token.

A station that seizes the token changes its mode from REPEAT to TRANSMIT. In TRANSMIT mode the station transmits its message in the Data field of the data-frame and waits for the frame to come back. Upon receiving the frame back, the station converts it from a data-frame into a token-frame by changing the token bit to 0 and stripping the Destination, Source and Data fields. In addition, if $P^- < \max(R_2^-, Pm)$ the station adapts the token priority to the highest reservation by setting $P^+ \leftarrow \max(R_2^-, Pm)$. If $P^- > \max(R_2^-, Pm)$, namely the highest reservation is less than the current priority, there is only one station that may decrease the priority to the highest reservation. This is *the last station to have sent a message using a token with priority less than or equal to* $\max(R_2^-, Pm)$. This rule ensures *fairness.*

Note that whether $P^- > \max(R_2^-, Pm)$ holds or not, the station performs $R_1^+ \leftarrow \max(R_2^-, Pm)$. This is because in a token-frame R_1, rather than R_2, indicates the highest reservation. This implies that a station in TRANSMIT mode does not operate with one-bit-delay. This fact, which holds for the original protocol as well, is meaningless because a token-frame never encounters a station in TRANSMIT mode, and a data-frame encounters only one station in this mode — its sender.

In the new protocol, changes in the priority level are not remembered in stacks. Stacks are not convenient here since if, for example, a station raises the priority from 1 to 3 and afterwards from 5 to 6, we allow the station to decrease it in one step from 6 or 7 not only to 5, but also to 1 or 2. Therefore, we use here a vector V with 8 three-bit entries (assuming there are 8 priority levels). The vector V at station i is denoted by V_i and its r'th entry by $V_i[r]$, where $0 \leq r \leq 7$. At initialization, for every r $V_i[r] \leftarrow 7$. Afterwards, when station i increases P from p to p', it performs $V_i[r] \leftarrow r$ for every $p \leq r < p'$ and $V_i[r] \leftarrow \min(p, V_i[r])$ for every $r < p$. For example, suppose that station i increases P from 1 to 3 and then from 5 to 6. After the first increase $V_i = (1,1,2,7,7,7,7,7)$, whereas after the second increase $V_i = (1,1,2,5,5,5,7,7)$. As shown later, entry $V_i[r]$ indicates the value station i should decrease P to, given that $R_1 = r$ and some additional conditions are satisfied.

The new priority mechanism can now be summarized as follows:

(a) A station can send a message with priority Pm only when it receives a token, provided that it does not change the priority field of this token and $P \leq Pm$ holds. A station that receives a token and alters P can only decrease it. Allowing such a station to seize the token and send its message immediately would result in an unfair scheme.

(b) A station makes a reservation for a token with priority level Pm by setting $R_1^+ \leftarrow \max(R_1^-, Pm)$ in every received frame. If the station detects later that $T^- = 1$, namely the received frame is

a data-frame, it makes a reservation in R_2 by setting $R_2^+ \leftarrow \max(R_2^-, Pm)$.

(c) A station that does not change P and recognizes that $P^- \leq Pm$ and $T^- = 0$ considers the received frame as a *usable token*. The station may seize the token by replacing it with a data-frame containing its own message. This is done by changing T from 0 to 1, resetting R_2 to 0 and appending Destination, Source and Data fields.

(d) A station that receives its data-frame back after one round-trip replaces it with a new token. The value of the priority field P is determined as follows. If $P^- < \max(R_2^-, Pm)$, then $P^+ \leftarrow \max(R_2^-, Pm)$. If $P^- = \max(R_2^-, Pm)$, then P does not change ($P^+ \leftarrow P^-$). If $P^- > \max(R_2^-, Pm)$, then P is decreased only if the conditions specified in (e) below are satisfied. In addition, the station sets $R_1^+ \leftarrow \max(R_2^-, Pm)$.

(e) A station that increases P from p to p' is responsible for decreasing it later to any value in the range $[p\ ,\ p'-1]$. Such a change can be performed only in a token ($T = 0$) with $R_1 < P$.

In order to illustrate rule (e) and the operation of the vector V that replaces the stacks of the IEEE-802.5 standard, suppose that station i increases P from 0 to 3, and then station j increases it from 3 to 7. As explained before, after this happens, $V_i = (0,1,2,7,7,7,7,7)$ and $V_j = (3,3,3,3,4,5,6,7)$. Suppose that a token with $0 \leq R_1 \leq 2$ and $P > R_1$ circulates around the ring. In this case, one of the following scenarios may take place:

- if the token is received by j first, j reduces P to 3, since $V_j[R_1] = 3$ holds for $0 \leq R_1 \leq 2$; afterwards, station i reduces P to the desired value R_1, since $V_i[R_1] = R_1$ holds for $0 \leq R_1 \leq 2$.
- if the token is received by i first, i reduces P *directly* to R_1, since $V_i[R_1] = R_1$ holds for $0 \leq R_1 \leq 2$.

However, if $R_1 < P$ but $3 \leq R_1 \leq 6$, only j can decrease P (directly to the required value). This is because in such a case $V_i[R_1] < P$ does not hold, but $V_j[R_1] = R_1 < P$ holds.

In order to illustrate the differences between the new approach and the one of IEEE-802.5, consider the following scenario. Suppose the priority P is 0, and station i is the first to seize the token and transmit a data-frame with a message. Suppose that when station i receives the frame back, it recognizes that the maximal reservation (R in the IEEE-802.5 protocol, R_2 in the new protocol) is 2. Therefore, in both mechanisms i increases P from 0 to 2. When the station that has reserved $R = 2$, j say, receives the token and recognizes that $P = 2$, it seizes the token and transmits a data-frame with its own message. Suppose that when the frame is received back, station j finds that the next highest reservation is for priority 4. Therefore, it increases P from 2 to 4, and the next station that has made the reservation for 4, k say, seizes the token and transmits a data-frame. When this station receives the frame back, it finds that the next reservation is for priority 6. Thus, it releases the token with $P = 6$, and the station that has made the reservation for 6, l say, seizes the token and transmits a frame.

So far all events in the new protocol are similar to those in the old one. However, suppose that when station l receives the frame back, it finds that the maximal reservation is for priority 0, namely no station holds a message whose priority is larger than 0. In the IEEE-802.5 protocol, only station k, which was *the last to have increased P to 6* can decrease the priority from 6. Then, j can decrease P from 4 to 2. Only when i recognizes that $P = 2$,

$T = 0$ and $R = 0$, it decreases P to 0. Depending on the location of i, j, k and l, this process may take up to 3 round-trips. In the general case, where the ring has $\mathcal{P}$ priority levels, this process takes up to $\mathcal{P}$ round-trips. In the new protocol, on the other hand, when station i receives the token for the first time (and recognizes that the maximal reservation is 0, but $P > 0$), it decreases P to $V_i[R_1] = 0$, whether or not the received value of P is 2. This is because i *was the last station to have increased* P *from 0.*

Next we describe the one-bit-delay algorithm for decreasing the priority of an unusable token, for which $T^- = 0$ and $T^+ = 0$ holds. At first glance, this seems to be an impossible task. This is because in order to know that a received frame is an unusable token, a station must completely receive P and T, but when T is received (or even when P, which is a 3-bit field, is completely received), it is too late to decide the value of P while working with one-bit-delay. Therefore, a stacking station must know *before* receiving P^- and T^- whether the received frame is an unusable token. This is the most interesting part of the new scheme.

A station is allowed to decrease the priority P to $V[R_1]$, where $V[R_1] < P$, if and only if it receives an unusable token, namely when $T^- = 0$ and $P^- > Pm$ holds. Note that this condition is mandatory in the IEEE-802.5 standard's protocol as well. The reason for requiring $T^- = 0$ is that when $T^- = 1$, the maximal reservation is unknown, since only part of the stations have had the opportunity to make a reservation. $P^- > Pm$ is required because $P^- \leq Pm$ indicates that the current service priority is still needed or even should be increased.

Upon receiving a frame, and after having updated R_1 ($R_1^+ \leftarrow \max(R_1^-, Pm)$), a station in REPEAT mode always *sets* P^+ *to the minimum of* $V[R_1^+]$ *and* P^- (as shown in Section 1, this can be done with one-bit-delay). We now explain why this action does not change P when $T^- = 1$ or when $T^- = 0$ and $P^- \leq Pm$.

First consider the case where $T^- = 1$ (the received frame is a *data-frame*). As proved in [3], $T^- = 1$ implies that (a) $R_1^- \geq P^-$. Since $R_1^+ \leftarrow \max(R_1^-, Pm)$, (b) $R_1^+ \geq R_1^-$ holds too. From (a) and (b) follows that (c) $R_1^+ \geq P^-$. In addition, (d) $V[r] \geq r$ holds for every station at any time. From (c) and (d) follows that $V[R_1^+] \geq P^-$. Thus, the operation $P^+ \leftarrow \min(V[R_1^+], P^-)$ does not change P.

Now consider the case where $T^- = 0$ and $P^- \leq Pm$ (the received frame is a *usable token*). Since $P^- \leq Pm$ and since $R_1^+ \leftarrow \max(R_1^-, Pm)$, it follows that $R_1^+ \geq P^-$. This follows similarly to (c) in the previous case and so is the rest of the proof.

4 Formal Specification

Table 1 presents the formal specification of the station algorithm. Recall that a station is usually in REPEAT mode, in which case it may seize a token or make a reservation. A station that seizes a token enters TRANSMIT mode and transmits its message. After transmitting the message, a station in TRANSMIT mode waits for its data-frame to come back. Then it converts it into a token-frame and returns to REPEAT mode.

Recall also that the superscript "–" denotes a value of a field in the received token or data-frame, whereas the superscript "+" denotes the value of a field in the transmitted frame. In TRANSMIT mode, R_2 has an intermediate value, $\max(R_2^-, Pm)$, which is determined after R_2^- is known, but is changed (to

- in REPEAT mode:

$R_1^+ \leftarrow \max(R_1^-,\ Pm)$
$P^+ \leftarrow \min(V[R_1^+],\ P^-)$
if $(P^+ \leq Pm) \wedge (P^+ = P^-) \wedge (T^- = 0)$ then (* the token is seized *)
$T^+ \leftarrow 1$
$R_2^+ \leftarrow 0$
change mode to TRANSMIT
transmit Destination, Source and Data fields
else $R_2^+ \leftarrow \max(R_2^-,\ Pm)$
update the local vector V (as shown below)

- in TRANSMIT mode:

$\tilde{R}_2 \leftarrow \max(R_2^-,\ Pm)$
$T^+ \leftarrow 0$; $R_1^+ \leftarrow \tilde{R}_2$, $R_2^+ \leftarrow 0$
if $\tilde{R}_2 > P^-$ then
$P^+ \leftarrow \tilde{R}_2$
$\forall r \in [0,\ P^- - 1]$ do $V[r] \leftarrow \min(V[r],\ P^-)$
$\forall r \in [P^-,\ P^+ - 1]$ do $V[r] \leftarrow r$
else $P^+ \leftarrow \min(V[R_1^+],\ P^-)$
change mode to REPEAT and update the local vector V (as shown below)

- Updating The Local Vector V

$\forall r \in [P^+,\ 7]$ do $V[r] \leftarrow 7$
$r \leftarrow P^+ - 1$
while $(r \geq 0) \wedge (V[r] \neq r)$ do
$V[r] \leftarrow 7$
$r \leftarrow r - 1$

Table 1: Station Algorithm Upon Receiving a Token- or a Data-Frame

0) when R_2^+ is transmitted. This value is occupied in a local variable, called $\tilde{R}_2$.

5 Main Properties of the New Mechanism

The present section outlines the main properties of the new mechanism, as stated and proved in [3].

- The algorithm in REPEAT mode can be performed with *one-bit-delay.*
- The protocol ensures liveness in the following sense: (a) within *at most 1 round-trip* (instead of up to $\mathcal{P}$ round-trips in the IEEE-802.5 standard) after a token is issued, either one of the stations seizes it and transmits its message, or the priority field P of the token contains the value of the maximal reservation; (b) the token cannot circulate the ring for more than 2 consecutive round-trips, (instead of $\mathcal{P} + 1$ in the standard protocol), if some messages are waiting for transmission.
- The protocol ensures fairness in the following sense: (a) messages with higher priority are sent first; (b) if station i seizes a token and transmits a waiting message with priority p at the time when station j has a waiting message with the same priority, then j will get an opportunity to transmit its message before i will

get another opportunity to transmit a message with priority p.

6 Conclusion

The paper has addressed the issue of station delay in ring networks. It has explained why one-bit-delay is the minimum possible delay at a ring station and shown that the station delay depends on the Medium Access Control protocol executed in the ring.

Then, the paper has introduced the distributed priority mechanism for token-rings as approved by the IEEE-802.5 standard. We have shown that due to the computation restrictions imposed by the one-bit-delay requirements, this mechanism leads to loss of bandwidth and starvation.

The paper has presented a new priority mechanism. The new mechanism retains all the desired properties of the standard: it can be executed by the ring stations with one-bit-delay and it ensures liveness and fairness. However, in the new protocol when $P > R$ holds, P is reduced to R in *at most 1 round-trip* rather than up to $\mathcal{P}$ round-trips. This minimizes the loss of bandwidth and enables low priority messages to be served much faster.

References

[1] W. Bux, F.H. Closs, K. Kuemmerle, H.J. Keller and H.R. Mueller, "Architecture and Design of a Reliable Token-Ring Network", *IEEE Journal on Selected Areas in Comm.*, Vol. SAC-1, No. 5, pp. 756-765, Nov. 1983.

[2] R. Cohen and A. Segall, "An Efficient Reliable Ring Protocol", *IEEE Transaction on Communications*, Vol. 39, No. 11 November 1991.

[3] R. Cohen and A. Segall, "An Efficient Priority Mechanism for Rings", T.R. 606, Dept. of Computer Science, Technion, IIT.

[4] Institute of Electrical and Electronics Engineers Inc. ANSI/IEEE Standards 802.5-1989 (ISO 8802-5), Token Ring Access Method and Physical Layer Specification".

[5] J. Pachl and L. Casey, "A robust ring transmission protocol", *Computer Networks and ISDN Systems*, 13 (1987), pp. 313-321.

[6] W. Stallings, "Fairness in Lans: Is the Performance Price Worth it?", *Data Communications*, February 1988, pp. 151-160.

Connection-Based Communication in Dynamic Networks

Extended Abstract

*Amir Herzberg**

Abstract

We analyze and improve the fault tolerance of practical, efficient end to end communication schemes. We concentrate on *connection-based source routing* schemes, used in most existing wide-area networks, e.g. in SNA/APPN. These schemes are composed of three components: a *topology update* protocol, a *route selection* algorithm and a *connection* protocol. The topology update protocol maintains an approximation of the network topology at every processor. The route selection algorithm in the source processor uses the topology approximation to select the 'best' route to the destination. The connection protocol sends messages along this route. We make the following contributions:

- Explicitly expressing the high efficiency possible with connection-based schemes. This efficiency is well appreciated in practice, but has not been analyzed formally so far. Roughly speaking, both communication and time complexities are in the order of the shortest route from source to destination which is up for 'long enough'.
- Understanding the tolerance of the scheme used in SNA/APPN. Roughly, this scheme operates efficiently if the mean time between failures (MTBF) of the network is at least the number of processors. The scheme may fail for lower MTBF even if a route from source to destination never fails.
- A new route selection algorithm which ensures efficient operation even when failures are frequent (low MTBF), as long as some route from source to destination is up for sufficiently long.
- Formalizing the notion of MTBF.
- A crash-tolerant, self-stabilized connection protocol with bounded storage and messages.

*IBM T.J. Watson, Yorktown Heights, NY 10598. E-mail: amir@watson.ibm.com

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0013...$1.50

1 Introduction

The usual form of communication in networks is *end to end*, i.e. data from a *source* processor is transmitted to a *destination* processor. We consider this task in the setting of point-to-point wide area networks based on packet switching, e.g. SNA [BGJ+85]. We are interested in the resiliency to failures. Formally, we use the *dynamic networks* model of [AE86], i.e. an asynchronous message-passing network $G(V, E)$ with general topology in which links fail and recover.

All practical end to end protocols are based on sending every packet along a single route from source to destination. The protocols differ in their strategies for selecting this route and ensuring FIFO and delivery. However, all protocols try to use a 'short' (i.e.

efficient) route.

We concentrate on protocols which are based on *connections* (also called *sessions* or virtual circuits). In such protocols, the communication is done by 'setting-up' an 'efficient' route between the source and the destination, and only then communicating along this route. This is efficient if much data is communicated over the connection, since most routing information can be sent just once during set-up, and not with each piece of data. If only one message is sent, it is more efficient to use 'connection-less' schemes.

We further concentrate on *source-routing* schemes, in which the source processor selects the route to the destination based on some topology data. Connection-based communication with source routing is widely used in practice, for example in SNA [BGJ+85, Gro82] and in Codex [BG87].

Connection based source routing schemes are composed of three components: a *topology update* protocol, a *route selection* algorithm and a *connection* protocol. The topology update protocol maintains an approximation of the network topology at every processor. The route selection algorithm in the source processor uses the topology approximation to select the 'best' route to the destination. The connection protocol sets up the route and then sends efficiently data messages along the route.

Our Contributions

Our main contribution is an explicit analysis of the high efficiency possible with connection-based schemes. Loosely speaking, the communication complexity of our modification of the SNA/APPN scheme is in the order of the 'shortest stable up' route from the source to the destination. More precisely, the average delay and the communication cost per packet sent are $O(l)$, where l is the length of the *shortest* $O(|E| \cdot l)$*–Up route* from source to destination. An $O(|E| \cdot l)$–Up route consists of links which are up for the last $O(|E| \cdot l)$ time units. This efficiency is well appreciated in practice, but has not been analyzed formally so far.

Additional Contributions

- Understanding the sensitivity to failures of the scheme used in SNA/APPN. SNA is the most common point to point network, and SNA/APPN is its newer version. Hence, the effects of failures on the scheme used in SNA/APPN are of great practical importance. However, while it is well understood that in practice SNA/APPN operates well in spite of some failures, this was not analyzed so far. We give precise sufficient and necessary conditions for efficient operation of the SNA/APPN routing scheme. In particular, we show that SNA/APPN requires mean time between failures (MTBF) of at least $|V|$, even if some route from source to destination never fails.

- A simple modification to the route selection algorithm, which allows the SNA/APPN scheme to operate efficiently whenever the source and the destination are connected by a route which is up for a period of length $\Omega(|E| \cdot |V|)$. The modified scheme also operates if the MTBF is $|V|$ or more.

- An self-stabilized, crash-tolerant connection protocol, which uses only bounded resources (without unbounded counters). The protocol is very simple and efficient. We present also a simpler version which is bounded and crash-tolerant (but not self stabilized). This version uses the same flows as the existing connection protocol of SNA/APPN [SJ86, BGJ+85], and therefore is very practical.

  Connection protocols in use today are neither self-stabilising nor crash-tolerant [SJ86, BGJ+85]. A self-stabilizing connection protocol was suggested in [Spi89, Spi90]. All of these protocols deal with the simple case where only one route is used between each source and destination. The only solution proposed to the more realistic case where many routes are used is unbounded counters or time-stamps [Gro82] (which is not self-stabilized).

A self-stabilized (crash-tolerant) end to end protocol is obtained by using our connection protocol together with a self-stabilized (crash-tolerant) topology update protocol, e.g. [Spi86, HS89]. Further work is required to analyze the complexities of the resulting scheme, however they seem to be inferior to the excellent complexity of the present scheme.

- A quantified definition of the MTBF of a link and a network. These concepts extend the quantitative approach as presented in [AGH90, GHS]. Our work gives additional evidence that this approach allows formal analysis with practical significance.

Related Works

Previous formal analysis of end to end protocols showed complexities which are a function of the size of the network, not of the length of the shortest 'possible' route from source to destination. Hence, these protocols appear much less efficient than the APPN protocol, especially with the improvement suggested in this work. We first compare this work to the protocols analyzed using the quantitative definition of a $f()$−Up route [AGH90, GHS]. Both of these works suggested 'connection-less' end to end protocols. However, these protocols are much less efficient than the one of this paper. The protocol of [GHS] has communication complexity of $O(|E|)$, and furthermore requires that the *every two processors* will be *always* connected by a $O(|V|)$−Up route. The protocol of [AGH90] has communication complexity $O(|E|)$ when amortized over prefixes of the execution. Furthermore, under the more realistic communication complexity measure of [GHS] (also Def. 3 below), which allows amortization over any sufficiently long interval, the complexity of the end to end in [AGH90] is exponential. Note that the *broadcast* protocol which is the main result of [AGH90, GHS] is efficient under both measures.

We now compare our assumptions to the two assumptions used in most existing formal works in this area. The *eventual stability* assumption [Gal76, Fin79, Seg83, AAG87] is that after some finite but unbounded time, there will be no more failures. All practical protocols work under this assumption. However, this assumption is insufficient to ensure any progress concurrently with failures. Hence, the complexity measure used in these works is the average complexity *per failure*, which does not capture the actual efficiency of connection-based schemes. In reality, failures may be frequent [RS91], and protocols are designed to be resilient to concurrent failures. The eventual stability assumption does not enable us to measure this resilience to concurrent failures.

The other assumption used in many existing formal works is that the network is *eventually connected*. Loosely speaking, two processors are *eventually connected* if there is a route between them with links that 'sometimes' operate [AE86]. Obviously, this is an extremely weak requirement. Unfortunately, it is so weak that reasonable solutions appear incorrect, and unreasonable impossibilities and lower bounds hold. In particular, to ensure communication between eventually connected processors we must try every possible route. This gives communication cost of $\Omega(|E|)$[1]. If one allows unbounded counters, then an $O(|E|)$ solution is by flooding [Vis83]. However, this is still too inefficient for practical use. Clearly, the much less efficient solutions without unbounded counters [AG88, AMS89, AG91, AGR92] are impractical.

2 Definitions

2.1 The Dynamic Networks Model

We consider the dynamic network model of [AAG87, AE86]. The network is represented by an undirected graph $G(V, E)$, with $n \stackrel{\text{def}}{=} |V|$ *processors* and $m \stackrel{\text{def}}{=} |E|$ *links*. Each link (u, v) enables direct communication between processor u and v; we say that u and v are *neighbors*. There is no assumption about the topology of the graph, and the topology is a-priori not known to the processors. However, the processors know n, and each processor has a distinct identity.

[1] However, [AGR92] show that if the data transmitted is very long, cost of $O(|V|)$ is possible!

The communication over each link satisfies the *data reliability* properties as defined in [BS88]. The main property is that FIFO is preserved. Most properties deal with periods of time when the link is 'up'. The link is *up* at a processor from a recovery until its next failure.

We assume total order between events. We also associate a positive number $time(e)$ with each event e. The number $time(e)$ represents the 'normalized time' of event e. Namely, the transmission delay is at most one time unit, regardless of the amount of current traffic. The use of the total order and of the time is only for the sake of analysis, and completely transparent to the protocol. In particular, the processors are not aware of the 'time' of events during the execution. Hence, the model is completely asynchronous.

2.2 Quantitative and Dynamic Link Properties

The main idea of the quantitative approach enunciated in [AGH90, GHS] is that for some purposes, it may be required that the link be up for some period continuously. This is easily expressed by the definition below of a link being τ−Up, which essentially refines[2] definition 2.1 from [AGH90].

Definition 1 *We say that link (u,v) is τ−Up at time t, if (u,v) is up at both v and u during the entire interval $[\max(t-\tau,0),t]$.*

Note that v knows, at any moment, if (u,v) is up at v. However, processor v does not know if (u,v) is currently τ−Up. One reason is that the network is asynchronous and hence v cannot detect when τ time units have elapsed. Another reason is that processor v does not know if (u,v) is up at u.

We now present a new definition which formalizes the *mean time between failures*, or MTBF, of a link or of the network at a given time period. It seems that many practical protocols are sensitive to the MTBF. Some networks even contain mechanisms to delay recoveries in order to ensure high MTBF [RS91].

Definition 2 *A link (u,v) has MTBF of μ during $[t_1,t_2]$ if the number of failures of (u,v) during $[t_1,t_2]$ is at most $\frac{t_2-t_1}{\mu}$. Similarly, the network has MTBF of μ during $[t_1,t_2]$ if the total number of failures during $[t_1,t_2]$ is at most $\frac{t_2-t_1}{\mu}$.*

2.3 Communication Complexity

We found it crucial to the understanding of practical protocols to use a definition of communication complexity that averages the communication over intervals with specific properties, for example long enough. This definition is taken from [GHS], and see there more elaborate motivations.

The communication complexity is defined with respect to a predicate P of intervals. Loosely speaking, C is the communication complexity of a protocol with respect to predicate P, if in every execution, the number of packets received by the protocol in every interval which satisfies P is at most C.

The function C depends on the size of the network and on the number of packets *accepted* for transmission from the higher layer during the interval. In dynamic networks, C depends also on the number of *failures and recoveries* during the interval.

Some or all of the packets sent and received during a interval may be due to packets accepted, failures and recoveries that have occurred *before* this interval. The communication complexity therefore considers also the events during a certain interval before the measured interval. To reduce notations, we require that both intervals are of the same length.

Definition 3 *We say that function C is the* communication complexity of the protocol over intervals satisfying *P, if in every execution, the number of receive events during any interval $(t,t+x]$ satisfying P is at most*

$$C(n,m,A,F,R)$$

where:

[2] The term τ−Up was suggested by David Peleg, instead of τ−Reliable in [AGH90].

$A \stackrel{\text{def}}{=}$ *The number of messages* Accepted *in the source during* $[t-x, t+x]$.

$R \stackrel{\text{def}}{=}$ *The number of* Recover *events during* $[t-x, t+x]$.

$F \stackrel{\text{def}}{=}$ *The number of* Fail *events during* $[t-x, t+x]$.

If $C(n, m, A, F, R) = A \cdot C_A(n, m) + F \cdot C_F(n, m)$ *then we say that the communication is* C_A *per accept and* C_F *per failure.*

Restricting the length of packets. The definition above counts only the number of events, without taking into account the size of the packets involved. To justify this, we assume that each packet may contain only one of the following: a processor identity, a $O(\log n)$ length field, and a 'counter'.

2.4 Throughput

Intuitively, the throughput is the rate at which the higher layer may send packets using the protocol. For simplicity, we assume that the higher layer always has packets to send. This definition is from [GHS], where it is given without this assumption.

Like communication complexity, the throughput is traditionally measured for the worst case prefix of an execution of the protocol. As for communication complexity, we see no apparent reason to consider only prefixes of executions, which may hide important transient effects. Instead, we propose to consider the throughput of any *interval* of an execution of the protocol.

Definition 4 *We say that function* T *is the* throughput of the protocol over intervals satisfying predicate P, *if in every execution, the number of packets delivered at the destination during any interval* $(t, t+x]$ *satisfying* P *is at least* $x \cdot T(n, m)$.

3 Analyzing and improving the SNA/APPN Scheme

Connection-based schemes are composed of three components: a topology broadcast protocol, a route selection strategy and a connection protocol. The *topology broadcast* protocol maintains topology databases in the processors, trying to keep them as close to the actual topology as possible. Whenever a *source* processor wishes to communicate with a *destination* processor, the *route selection* strategy in the source uses its topology database to select the 'best' route to the destination. Then, the *connection protocol* tries to send messages over this route to the destination. For efficient transmission of many messages over the route, the connection protocol sends the description of the route only in the first few control messages, used to *set up* the route. The route selection strategy may also pick a new route because of failure or otherwise. Then, the connection protocol takes down the old route and sets up the new route.

3.1 The SNA/APPN Scheme

In this subsection, we investigate the communication scheme used in SNA/APPN [BGJ+85, JBS86]. Like other practical schemes, this scheme is based on the use of 'unbounded counters' to identify the order between packets. This greatly simplifies the implementation of the topology broadcast and connection protocols.

The topology update protocol in APPN is based on *flooding*. Each topology change, once detected, is sent to all neighbors with a sequence number and the identity of the processor that detected the topology change. Whenever a processor receives a packet describing some topology change detected by another processor, it compares the sequence number in this packet to the last sequence number it received from that processor. If the packet received contains a smaller or equal number to the one stored, then this packet is ignored (being old). Otherwise, the packet is sent to all neighbors, the topology estimate is updated and the sequence number of changes from that processor is updated.

This simple protocol ensures that the source will be updated about the state of every processor connected to it by a route which is up for 'long enough', as formalized in the theorem below.

Theorem 1 *There is a topology update protocol with communication complexity $O(m)$ per change for intervals of length n and with the following property. If processors u and v are connected by an $(l+2)-Up$ route of length l at time t, then the state of the links of v declared by the protocol in u at time t is the actual state in v at some time during $[t-(l+2),t]$. Furthermore, the number of topology changes in any link declared by the protocol in u during $[t-\tau,t]$ is at most the number of actual changes during $[t-(l+2)-\tau,t]$.*

Proof: See algorithm 2.3b of [JBS86]. □

The estimate produced by the topology update protocol is used in APPN to select the shortest operating route to the destination. Whenever a shorter route becomes operating, then it is used instead.[3]

The selected route is used by the connection protocol to send the data messages. Each message is sent in a packet, encapsulated together with control information which allows the packet to be forwarded to the destination. This control information should be short in order to ensure efficient communication. This is done by sending the description of the route only with a set-up packet at the beginning of the use of the route. When an intermediate processor receives the set-up packet, it stores the identity of the next neighbor toward the destination, and an identifier of this connection. After this set-up process, each data packet contains only this identifier. APPN uses the set-up procedure of [SJ86], which allows a short identifier, and requires $2l$ to complete where l is the length of the route.

Unbounded counters are used only in the set-up process. The source appends a connection counter, which is used to discard old connections in favor of newer connections [Gro82]. In the next section this unbounded counter is eliminated.

While sending data packets, APPN uses a *window*, i.e. allows several data packets to be forwarded along the route at the same time. This ensures high throughput ($\Omega(1)$) as long as the same route is used. For simplicity, assume every hop (link) contains at most one packet. Below we summarize the properties of the connection protocol used in APPN.

Theorem 2 *There is a connection protocol which ensures that messages are always delivered in the order in which they were received (FIFO). Also, assume that during $[t,t+x]$, a specific route of length l s.t. $x > 6l$ is selected for this connection protocol. Then the communication is at most $3l$ per accept and l^2 per failure of a link in this route. If the route is up during $[t,t+x]$, then the throughput is $\Omega(1)$ during $[t,t+x]$. Both complexities are for intervals of length $6l$.*

Proof Sketch: The FIFO follows from [SJ86]. If the route does not fail, the set-up costs are amortized and negligible. If the route fails, the costs are attributed to the failures. □

As mentioned above, APPN uses topology update according to Theorem 1, selects always the shortest route, and uses the connection protocol of Theorem 2.

Theorem 3 *Consider a connection-based communication scheme where the shortest route is always selected and with topology update and connection protocols satisfying Theorems 1 and 2, respectively. Then the communication complexity is $O(l)$ per message accepted and $O(m)$ per failure, and the throughput is $O(1)$, for intervals $(t,t+x]$ satisfying:* (a) $x \geq 24 \cdot l \cdot m$, (b) *during $[t,t+x]$ the MTBF of the network is at least $6l$ and* (c) *during $[t-2n,t+x]$ there is a $3l-Up$ route of length l or less from source to destination.*

Proof Sketch: Since the shortest route is always used, then from Theorem 1 all routes used during $[t-n,t+x]$ are of length l or less. Since the MTBF during $[t,t+x]$ is at least $6l$, it follows that there would be at most $\frac{x}{6l}$ failures during $(t,t+x]$. Hence there would be at most $\frac{x}{6l}+m$ recoveries. Since routes are switched only upon a failure or a recovery, there are at most $2 \cdot \frac{x}{6l}+m$ route switchings during $(t,t+x]$.

[3] In some actual implementations of APPN, a route is used until it fails. This may cause the use of an inefficient route until it fails. We analyse above the improvement of [Gro82], where the shortest route is always used (if necessary, taking down intentionally the operating route). The only advantage we found for the other method is that it is robust to frequent recoveries.

Hence, from Theorem 2 and simple arithmetics, the number of messages delivered is at least $\frac{x}{12} - ml$, from which the throughput follows.

To compute the communication complexity, we consider only the connection protocol since Theorem 1 already shows that the topology update contributes at most $O(m)$ per failure. Furthermore, we are concerned only with the set-up and take-down costs which are both at most $O(l^2)$, since successful transmissions require only l send events. As argued above, there are at most $2 \cdot \frac{x}{6l} + m$ route switchings during $(t, t+x]$, and at least $\frac{x}{12} - ml$ deliveries. The claim follows by arithmetics. □

3.2 Improving the route selection

Theorem 3 shows that APPN 'operates well' if the MTBF is $6l$ or more. If the MTBF is substantially smaller, e.g. l, then APPN does not work *even if there is a route which never fails* from source to destination with length l. This is possible even if each link fails only rarely. In this concise version we omit the details of this (simple) construction. We now present a simple modification to the route selection, which ensures operation whenever the source and the destination are connected by a route which is up for long enough.

The kind of scenario we wish to avoid is selecting alternately one of a pair of short but unstable routes, while a longer route is operating all the time. On the other hand, we should use a short route that has recovered and is up for 'long enough'.

A simple solution is to try the routes by order of increasing length, without trying again a link that failed. After trying all possible routes, or after trying for 'long enough', then we re-start the process from the shortest route.

Suppose there is a route r which is operating all the time, with length l. We try at most m routes before r, since we never try a link that failed. If the route r is operating, we use it to deliver 'enough' messages to compensate for the time and communication spent on trying other routes, in order to ensure good averaged complexities. The time spent is at most ml, and we try to get amortized throughput of 1; the communication spent is at most ml^2, and we try to get amortized l. Hence, it is sufficient to ensure that the total number of deliveries since last re-start with the shortest route is at most ml.

To summarize:

1. The algorithm starts a 'cycle' by trying to use the shortest route which is up according to the topology data.
2. Whenever the route in use fails, we use the shortest route whose links did not fail during this cycle.
3. We start a new cycle in one of the following cases:
   - when all routes from source to destination contain a link that failed in the current cycle, or
   - after delivering ml messages during a cycle, where l is the maximal length of a route used in this cycle.

This scheme works under very similar conditions to these in Theorem 3; we omit the proof. Furthermore, it also works when there is a sufficiently-up route from source to destination, without bounding the MTBF. This is shown in the following theorem.

Theorem 4 *Consider a communication scheme where routes are selected as described above and with topology update and connections protocols satisfying Theorems 1 and 2, respectively. This scheme has the properties of Theorem 3. Furthermore, the communication complexity is $O(l)$ per message accepted and $O(m)$ per failure, the throughput is $\Omega(1)$ and the average delay is $O(l)$. All the complexities are for intervals $(t, t+x]$ s.t. $x \geq 6 \cdot l \cdot m$ and during $[t-6nm, t+x]$ there is a route of length l or less from source to destination which is $6lm-Up$.*

Proof Sketch: A new cycle would begin at some time $t' \in [t - 6nm, t]$. By induction, every route selected during $[t', t + x]$ will be at most of length l. The complexities follow by arithmetics and Theorems 1 and 2. □

4 Connection Protocol

Our goal is to provide more robust implementations of connection protocols satisfying conditions similar to Theorem 2. In the first subsection we present a connection protocol which does not use unbounded resources (for counters). We then show that this protocol tolerates crashes. The third subsection contains the self-stabilized extension.

4.1 A Connection Protocol with Bounded Resources

The use of unbounded resources in existing connection protocols [Gro82] is only during the set-up phase. In the set-up phase, each intermediate processor stores the identity of the next processor along the route, so that the data packets sent later do not have to contain the description of the entire route. We now show a new set-up phase, which does not use unbounded counters. First, it is useful to understand why are unbounded counters used in [Gro82].

The unbounded counters are used in [Gro82] to distinguish between old and new connections between the same source and destination. Indeed works which consider only one connection do not use unbounded counters [SJ86, Spi89, Spi90]. There are two motivations for distinguishing between old and new connections:

Storage: Intermediate processors have to keep track of all the connections flowing through them. If the storage in an intermediate processor is bounded, then it must be able to use this storage for new connections rather than wasting it on old connections.

FIFO: The destination should preserve FIFO, and therefore it should never deliver messages from old connections.

The use of unbounded counters, given sequentially to the connections, enable the identification of a new connection by simply comparing the number of the connection to the largest connection number known. We now show alternative methods of addressing the two motivations above, without unbounded counters.

The simpler solution is for the storage (of intermediate processors). The solution is to allow at most one connection over each link, for each pair of source and destination. If a processor receives a set-up packet for a link which is already allocated to a connection with the same source and destination, then it delays this packet until one of the two connections is taken-down. When a connection is taken down, by the source or by a failure, then a 'take-down' packet propagates all over the route. This packet enables the intermediate processors to free the storage allocated to this connection.

The crucial point is that at most $O(n)$ time is required to wait for the old connections to be taken down. The time required is the time for detection of failures (which is assumed to be one time unit) plus the propagation time of the take-down packets along the route (one time unit per link). In fact, if during the last n time units all the routes used have been of length l or less, then the time is only $O(l)$. This is gives the efficiency required by Theorem 2 and thereby to implement Theorem 4.

We now consider the other motivation, namely preserving FIFO order among the messages delivered. This problem concerns only the destination. In this case, the destination receives the set-up packets of a connection. Before delivering messages from this connection, the destination should verify that this connection is indeed newer than the last connection from which the destination delivered messages. This is since, due to failures and uneven delays, an older connection may be received at the destination only *after* a newer connection. We have to test the 'freshness' of a connection when its set-up packet arrives at the destination.

The destination tests freshness of a route by sending along it, to the source, another control packet called *route-OK*. The source ignores route-OK from old connections (we later show how to identify route-OK packets which are from old connections that use the existing route). When the source receives route-OK from the *existing* connection, it starts sending

along the route the data packets. The data packets are forwarded toward the destination, unless the route is taken down (then they are discarded).[4]

We now explain how the destination can determine if the data packets are new. Consider two routes for which corresponding set-up, route-OK and data packets were received. Theorem 5 below shows that the order of the routes in the source is the same as the order between the time when the corresponding set-up packets were received at the destination. This is illustrated in Fig. 1. Note that the theorem holds both for the destination and for any intermediate processor v. Hence, intermediate processors may also use this indication of freshness, although it is not necessary (see above).

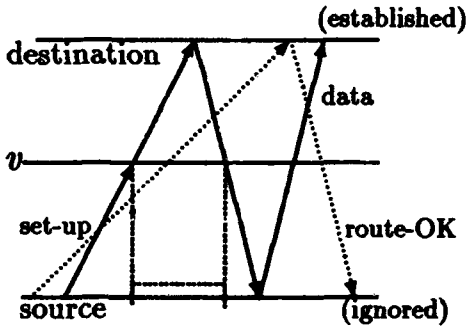


Figure 1: Possible timing of control packets

Theorem 5 *Assume that processor v received corresponding set-up, route-OK and data packets of two routes r, r'. Then the order between the times when the source sent the two set-up packets is the same as the order between the times at which v received them.*

The protocol above requires the source to identify the route-OK packet which was sent in response to the last set-up. We call a route-OK packet *stale* if it was *not* sent by the destination upon receiving the last set-up packet sent by the source. Our goal is to allow the source to ignore stale route-OK packets.

[4]The protocol uses exactly the same flows as the connection set-up protocol of [SJ86], which is used in SNA/APPN [BGJ+85] by appending unbounded counters [Gro82]. We conjecture, and hope to prove, that indeed the same actual flows could serve both purposes.

By simply including the route in the route-OK packet, the source can identify a stale packet, if each connection uses a different route. We now show how to enable the source to identify a stale route-OK packet, even if the same route was used before. The problem is that when the source issues the set-up packet, stale route-OK packets may be on transit on the same route, and they should be ignored. We solve this by ensuring the following:

Invariant: *every* processor ignores, and does not forward, any stale route-OK packet after it receives a newer set-up packet for the same route.

This invariant is trivially kept by the destination. To keep the invariant by all processors, we introduce another control packet, *local-ack*. Every processor sends local-ack immediately after receiving a set-up packet, to the neighbor which sent the set-up packet. Clearly, a route-OK packet received before the local-ack to the last set-up must be stale and is ignored. If the link fails and the set-up or local-ack packets are thus lost, then the route is taken down from both ends and any subsequent route-OK will be irrelevant. By induction from the destination, a stale route-OK cannot be received after the local-ack to the last set-up. Therefore, the local-ack mechanism implements the invariant.

4.2 Crash Tolerance

Processor crashes are failures where the processor loses its memory and re-starts from some predefined initial state [BS88]. The crash of the processor also causes the failure of all of its communication links. We now show that the response of the protocol above to these link failures is sufficient to handle the crash. Our discussion assumes that the data-link is crash-tolerant, namely that all of the properties of the link hold as if the crash was a link failure. This may be achieved if one bit per link is non-volatile [BS88] or by explicitly allowing probability of error [GHM89].

An obvious response to a crash in any processor is that all the connections flowing through that processor are taken-down and restarted. This holds also for the source and destination processors of a connection.

Hence, crash-tolerance involves only the set-up and the take-down phases of the protocol. Furthermore, no special function is required for the take-down, as it follows immediately from the link failure. It is easy to verify that the connection protocol using the set-up phase described above is also tolerant to crashes, with only the obvious response above. Note that there is no danger for confusion between connection set-up following the recovery and 'old' connections. The reason is that after the crash all the 'old' connections are taken-down.

4.3 A Self-Stabilized Connection Protocol

For discussion and definitions of self-stabilized protocols, see e.g. [KP90]. Our solution assumes that the data link is self-stabilized, e.g. as in [AB89], and delivers at most a constant number of packets before stabilization. We also assume that there is a periodic *time-out* event in every processor. For simplicity, when analysing the time complexity (only) we assume that at most one time unit passes between any two consecutive time-out events in any processor. This periodic event is required for self-stabilized protocols in the message-passing model, as shown in [KP90]. Intuitively, this is since a processor may be started in a mode where it just sent a packet and expects a reply, but the packet was never actually sent.

Our solution is that the source periodically sends a *check* packet on the route to the destination being set-up or used now. The check packet contains the description of the route. This enables the processors along the route to check that the tables are set properly (after set-up was completed), or to forward the packet (during set-up). The check packet also includes the previous packet sent along the route, which is compared to the previous packet received; they should be the same since the link is FIFO. If any error is found, the route is taken down. In particular, the route is taken down if it is not valid, e.g. if it has already been taken down. If no error occurs, the check packet arrives at the destination.

The destination also sends periodically a *check-back* packet to the source. This packet, again, contains the route and the previous packet sent, for the same purposes. Note that this packet is not really an acknowledgment to the check packet, e.g. the source cannot check if it received the expected response before the check-back arrives. The reason is that multiple check and check-back packets may be on transit at any moment, and the source cannot identify the check-back packet corresponding to the last check packet (except randomly as in [GHM89]).

The check packet is sent not only every 'time-out' period, but also every n data packets. This limits the number of data packets which are incorrectly routed. Intermediate processors verify that a check packet is received once every n packets, and if it is not received then the connection is taken down. This removes loops in the routing tables due to a transient error. Note that the amortized complexities are preserved.

A transient error may also cause a processor to have a 'bogus' connection, i.e. this connection is not defined in the other processors along the route. This may cause this processor to delay the newest set-up packet forever, waiting for the newest route or the bogus route to go down. To solve this problem, each processor periodically compares its active and waiting connections with its 'upstream' neighbors. Namely, processor u sends to neighbor v the description of its connection where u immediately follows v. This enables v to take-down any such bogus connection in the tables of u. If the tables of v also contain the bogus connection, it will be detected and taken down by the first processor upstream along the route with correct tables, or ultimately by the source. This process requires $O(n)$ communication for every link of the processor with the 'time-out' event, per each pair of source and destination. Note that with high probability this can be reduced to $O(1)$ for *all* source-destination pairs together, by using random hashing.

Theorem 6 *The connection protocol described in this section satisfies Theorem 2. Furthermore, for any execution with link failures, processor crashes and arbitrary initial state of the network at time* 0, *the following holds. The sequence of messages delivered*

after $O(n)$ is a prefix of the sequence of messages sent after $O(n)$. Let l be the (maximal) length of the route used. The communication after $O(n)$ is $O(l)$ per message, $O(l^2)$ per failure, and $O(n^2)$ per periodical wakeup. Also, assume that during $[t, t+x]$, for $t > 2n$, a specific route of length at most $\frac{x}{7}$ is selected and is up, and that during $[t-n, t]$ all routes selected were of length $\frac{x}{7}$ or less. Then the throughput during $[t, t+x]$ is $\Omega(1)$. All complexities are for intervals of length $O(l)$.

5 Conclusions and Open Questions

We analyzed a simple protocol for end to end communication in dynamic networks. This protocol is extremely efficient under the following quantitative condition: during an interval of length $O(nm)$, the source and the destination are connected by a route of length l which is $O(lm)$–Up. The complexities achieved are $O(l)$, which compares favorably with e.g. with the simple lower bound of $\Omega(m)$ on the communication complexity possible assuming only eventual connectivity. Is it possible to require only a shorter interval or allow a route which is up for less time? Can we find tradeoffs with the MTBF?

The result above requires unbounded counters. Can we achieve the above with bounded resources? We have shown an implementation of a bounded connection protocol. Hence, it is enough to find a bounded implementation of topology update with the properties of Theorem 1.

Acknowledgements

Thanks to Rafail Ostrovsky for helping to submit this abstract. Thanks to Hagit Attiya, Tsipi Barzilai, Oded Goldreich, George Grover, Shay Kutten, Yishay Mansour, Mike Merritt, Frank Schaffa, Adrian Segal, Gadi Taubenfeld and Moti Yung, for helpful comments and discussions. Moti also gave me [RS91].

References

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In 28^{th} *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[AB89] Y. Afek and G. M. Brown. Self-stabilization of the alternating-bit protocol. In *Proc. 8^{th} IEEE Symp. on Reliable Distributed Systems*, pages 10–12, 1989.

[AE86] Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.

[AG88] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 131–148. ACM SIGACT and SIGOPS, ACM, 1988.

[AG91] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proceedings of the 10^{th} Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 295–307. ACM SIGACT and SIGOPS, ACM, August 1991.

[AGH90] Baruch Awerbuch, Oded Goldreich, and Amir Herzberg. A quantitative approach to dynamic networks. In *Proceedings of the 9^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 189–204, August 1990.

[AGR92] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. To be presented in the eleventh PODC, 1992.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. Polynomial end-to-end communication. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science*, pages 358–363, October 1989.

[BG87] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[BGJ+85] A. E. Barats, J. P. Gray, P. E. Green Jr., J. M. Jaffe, and D. P. Pozefsky. SNA networks of small systems. *IEEE Journal on Selected Areas in Comm.*, SAC-3(3):416–426, May 1985.

[BS88] Alan E. Baratz and Adrian Segall. Reliable link initialization procedures. *IEEE Trans. on Communication*, COM-36:144–152, February 1988.

[Fin79] S. G. Finn. Resynch procedures and a failsafe network protocol. *IEEE Tran. Comm.*, COM-27(6):840–846, June 1979.

[Gal76] Robert G. Gallager. A shortest path routing algorithm with automatic resynch. unpublished note, March 1976.

[GHM89] Oded Goldreich, Amir Herzberg, and Yishay Mansour. Source to destination communication in the presence of faults. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 85–102, August 1989.

[GHS] Oded Goldreich, Amir Herzberg, and Adrian Segall. Quantitative analysis of dynamic network protocols. In preparation. The results are contained in the dissertation 'Computer Networks in the Presence of Faults' by Amir Herzberg, Computer Science Dept., Technion (in Hebrew, 1991).

[Gro82] George Grover. High availability for networks (HAPN) - nondisruptive VR switching. Internal memorandum, IBM T.J. Watson Research Center, August 1982.

[HS89] Pierre A. Humblet and Stuart R. Soloway. Topology broadcast algorithms. *Computer Networks and ISDN Systems*, pages 179–186, 1989.

[JBS86] Jeff Jaffe, Alan E. Baratz, and Adrian Segall. Subtle design issues in the implementation of distributed dynamic routing algorithms. *Computer networks and ISDN systems*, 12(3):147–158, 1986.

[KP90] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–102, 1990.

[RS91] T. L. Rodeheffer and M. D. Schroeder. Automatic reconfiguration in autonet. In *Symp. on Principles of Operating Systems*, 1991.

[Seg83] A. Segall. Distributed network protocols. *IEEE Trans. on Information Theory*, IT-29(1), January 1983.

[SJ86] Adrian Segall and Jeffe M. Jaffe. Route setup with local identifiers. *IEEE Trans. Comm.*, 34(1):45–53, January 1986.

[Spi86] John M. Spinelli. Broadcasting topology and routing information in computer networks. Technical Report LIDS-P-1543, MIT Lab. for Information and Decision Systems, March 1986.

[Spi89] J. M. Spinelli. *Reliable Data Communication in Faulty Computer Networks.* PhD thesis, MIT Lab. for Information and Decision Systems, 1989. Report LIDS-TH-1882.

[Spi90] J. M. Spinelli. Self-stabilizing network communication protocols. In E. Arikan, editor, *Proc. Bilkent International Conference on New Trends in Communication, Control, and Signal Processing*, pages 141–147. Elsevier Science Publishers, July 1990.

[Vis83] Uzi Vishkin. A distributed orientation algorithm. *IEEE Trans. Info. Theory*, June 1983.

Requirements for Deadlock-Free, Adaptive Packet Routing

Robert Cypher*

*IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Luis Gravano*†

†CRAAG
IBM Argentina
Buenos Aires, Argentina

Abstract

This paper studies the problem of deadlock-free packet routing in parallel and distributed architectures. We present three main results. First, we show that the standard technique of ordering the queues so that every packet always has the possibility of moving to a higher ordered queue is not necessary for deadlock-freedom. Second, we show that every deadlock-free, adaptive packet routing algorithm can be restricted, by limiting the adaptivity available, to obtain an oblivious algorithm which is also deadlock-free. Third, we show that any packet routing algorithm for a cycle or torus network which is free of deadlock and which uses only minimal length paths must require at least three queues in some node. This matches the known upper bound of three queues per node for deadlock-free, minimal packet routing on cycle and torus networks.

1 Introduction

This paper studies the problem of deadlock-free packet routing in parallel and distributed architectures. A wide range of packet routing algorithms with differing properties and costs have been proposed [1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 13, 14, 15, 16, 17, 19, 20]. In this paper we will focus on a particularly simple and important class of routing algorithms which we will call *queue-reservation* algorithms. A queue-reservation algorithm consists of rules that specify to which queues a packet may move based solely on the queue currently holding the packet, the packet's source node, and the packet's destination node. A packet is allowed to move from its current queue to any other queue at any time, provided that the other queue is empty and that the move is allowed by the routing algorithm. Queue-reservation algorithms can be implemented efficiently in hardware as they require only local information to make routing decisions, they are inherently asynchronous and therefore do not require a global clock, and they do not require the creation or exchange of any special packets containing only control information. Furthermore, adaptive queue-reservation algorithms allow packets to avoid congestion, thus permitting high throughput in the network. As a result of these advantages, queue-reservation algorithms have been widely studied and implemented.

The primary disadvantage of queue-reservation techniques is that they require that each node contain some minimum number of queues. Although a great deal of research has been devoted to the problem of minimizing the storage requirements of queue-reservation algorithms [2, 4, 5, 6, 8, 10, 12, 15, 17, 18, 19, 20], very little is known in terms of lower bounds on the storage which is required by such algorithms. Our goal in this paper is to characterize the properties which these algorithms must have in order to be free of deadlock and to use these properties to prove lower bounds on storage requirements.

One well-known technique for proving freedom from deadlock is to order the queues so that every packet

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0025...$1.50

always has the possibility of moving to a higher ordered queue [8]. Providing such an ordering of the queues is the standard technique for proving freedom from deadlock and has been used by many researchers [2, 4, 5, 6, 8, 10, 12, 15, 17, 19, 20]. Therefore, it seems plausible that the existence of such an ordering of the queues is a necessary condition for freedom from deadlock. In fact, in the special case of oblivious queue-reservation algorithms, Toueg and Steiglitz have shown that the existence of such an ordering of the queues *is* necessary for deadlock-freedom [18]. However, in this paper we will present an adaptive queue-reservation algorithm which is provably free of deadlock and for which no ordering of the queues can be defined such that every packet always has the possibility of moving to a higher ordered queue. Thus, in the case of adaptive routing the technique of ordering the queues is sufficient but not necessary for avoiding deadlock.

On the other hand, we will prove that every deadlock-free, adaptive queue-reservation algorithm can be restricted, by limiting the adaptivity available, to obtain an oblivious algorithm which is also deadlock-free. As a result, we will be able to use lower bounds on the storage requirements of oblivious routing algorithms to obtain lower bounds on the storage requirements of adaptive routing algorithms. In particular, we will show that any adaptive queue-reservation algorithm for a cycle or torus network which is free of deadlock and which uses only minimal length paths must require at least three queues in some node. This matches the known upper bound of three queues per node for deadlock-free, minimal routing on cycle [7] and torus networks [2].

The remainder of this paper is organized as follows. Definitions and a formal description of the routing model are given in Section 2. Section 3 presents an example of a deadlock-free, adaptive routing algorithm in which it is impossible to order the queues so that every packet always has the possibility of moving to a higher ordered queue. The fact that every deadlock-free, adaptive queue-reservation algorithm can be restricted to obtain a deadlock-free, oblivious algorithm is proven in Section 4. Lower bounds on the storage requirements for deadlock-free minimal queue-reservation algorithms are given in Section 5.

2 Preliminaries

We will view a routing network as being an undirected graph in which the nodes represent processors and the edges represent communication links. Each node contains a set of queues, one of which will be called an *injection queue*, another one of which will be called a *delivery queue*, and the remainder of which will be called *standard queues*. Packets can enter the routing network only by being placed in an empty injection queue in their source node, and they can be removed from the network only when they are in the delivery queue of their destination node. We will assume throughout that each queue can hold exactly one packet and that the number of queues is finite.

Given the queue in which a packet is currently stored, and given the packet's source and destination nodes, a *routing algorithm* specifies a set of queues to which the packet may be moved. More formally, the *color* of a packet is the pair (s, d) where s is the packet's source node and d is the packet's destination node. We will say that a queue has color c if it contains a packet with color c. A routing algorithm A is a function which associates a set of queues, called a *waiting set*, with each possible queue and color pair (q, c). The waiting set which A associates with the pair (q, c) will be denoted $A(q, c)$. Given a routing algorithm A, a queue q is *reachable* by a packet p with color c if and only if there exists some path $q_0, q_1, \ldots, q_k$ such that q_0 is the injection queue in p's source node, $q_k = q$, and for all i, $1 \leq i \leq k$, $q_i \in A(q_{i-1}, c)$. It is required that the waiting set $A(q, c)$ be empty if and only if either q is a delivery queue or q is not reachable by a packet with color c.

All of the queues in a waiting set $A(q, c)$ must either be in the node which contains q or in neighboring nodes (that is, nodes that are connected by an edge to the node containing q). An injection queue is never allowed to appear in a waiting set, and a delivery queue must only be reachable by packets destined for the node containing the delivery queue. Furthermore, if $q_2 \in A(q_1, c)$

and if either q_1 is an injection queue or q_2 is a delivery queue, then q_1 and q_2 must be in the same node. Thus injection and delivery queues are used only for placing new packets in the network and for removing packets once they have reached their destination [1].

The routing algorithm operates asynchronously. A packet with color c may move from a queue q_1 to any empty queue $q_2 \in A(q_1, c)$ at any time, and a new packet with an arbitrary destination may be placed in an empty injection queue at any time. Packets may be transmitted in either store-and-forward [15] or virtual cut-through [11] mode. The only requirement is that when a packet is moved from one queue to another, it occupies both of the queues for a finite amount of time, and after a finite amount of time the former queue becomes empty.

A routing algorithm is *oblivious* if every waiting set contains at most one queue, and it is *adaptive* otherwise. Routing algorithm A is a *restriction* of routing algorithm B if and only if for every pair (q, c), $A(q, c) \subseteq B(q, c)$, and for some pair (q, c), $A(q, c) \neq B(q, c)$. A routing algorithm is *minimal* if every packet is routed from its source node to its destination node while visiting the minimum number of nodes possible. Note that the concept of minimality is based on the number of nodes visited, rather than the number of queues visited.

A *configuration* is a nonempty set S of queues such that each queue q in S is either empty or has color c where q is reachable by a packet with color c. The set of queues S will be called the *critical set* of the configuration. Given a configuration T with critical set S and given any queue $q \in S$, the notation $T(q)$ will denote q's color in configuration T (or the value "empty" if it does not contain a packet). A *deadlock configuration* for a routing algorithm A is a configuration with a critical set S such that none of the queues in S is a delivery queue, none of the queues in S is empty, and for each queue q in S, q has color c where $A(q, c) \subseteq S$. A configuration is *routable* if and only if it is possible to start with an empty network and to route packets so as to obtain the configuration.

[1] It should be noted that the injection and delivery queues are introduced only to simplify the description of the model, and that they need not be physically present in an actual routing network.

A routing algorithm is *deadlock-free* if and only if it has no routable deadlock configuration. It is straightforward to verify that this definition of deadlock-freedom does in fact correspond to the impossibility of obtaining deadlock when using the given routing algorithm. Finally, given any two configurations T' and T'' with critical sets S' and S'', respectively, $T' \oplus T''$ will denote the configuration T with critical set $S = S' \cup S''$ in which for each queue $q \in S'$, $T(q) = T'(q)$, and for each queue $q \in S'' \setminus S'$, $T(q) = T''(q)$. Thus $T' \oplus T''$ is obtained by taking configuration T' and adding to it all of those queues in T'' which are not also in T'.

3 Deadlock-Freedom Without Ordering Queues

In this section we will show that the standard technique of ordering the queues so that every packet always has the possibility of moving to a higher ordered queue is not necessary for the prevention of deadlock. In particular, we will give a simple example of an adaptive routing algorithm which is provably free of deadlock and yet has no such ordering of the queues.

The example is routing algorithm A shown in Figure 1, in which each circle represents a queue and each arc represents a possible move between queues. There are three injection queues labeled I_1, I_2 and I_3, and three delivery queues labeled D_1, D_2 and D_3. In addition, there are six standard queues labeled X_1, X_2, X_3, Y_1, Y_2 and Y_3. We will consider only three colors of packets, namely C_1, C_2 and C_3, where packets with color C_i, $1 \leq i \leq 3$, are injected in queue I_i and delivered from queue D_i. The label associated with each arc specifies which color packets are allowed to make the given move between queues. For example, $A(I_1, C_1) = \{X_1\}$, $A(X_1, C_1) = \{X_2, Y_1\}$, and $A(X_1, C_2) = \{X_3\}$. Of course a complete routing algorithm would provide routes for packets with other colors and would include an assignment of the queues to the nodes in a routing network. However, it is straightforward to extend the given example by adding additional queues for the packets with other colors and to assign the queues to nodes in a routing network without

changing the deadlock or queue ordering properties of the example.

Lemma 3.1 *The routing algorithm A shown in Figure 1 is free of deadlock.*

Proof: Assume for the sake of contradiction that deadlock is possible, in which case there must be some routable deadlock configuration with a nonempty critical set S. Clearly, the delivery queues cannot appear in S. Similarly, Y_1, Y_2 and Y_3 cannot appear in S because they are only reachable by packets which are able to move directly to a delivery queue. Also, note that if injection queue I_i, $1 \leq i \leq 3$, is in S, then queue X_i must also be in S. Therefore, at least one of the queues X_i must be in S. Because none of the queues Y_i is in S, it follows that if X_1 is in S it must have color C_2 in the deadlock configuration, if X_2 is in S it must have color C_1 in the deadlock configuration, and if X_3 is in S it must have color C_3 in the deadlock configuration. Note that X_3 must be in S, because otherwise either X_1 or X_2 must be in S, and $X_3 \in A(X_1, C_2)$ and $X_3 \in A(X_2, C_1)$. Because $X_1 \in A(X_3, C_3)$ and $X_2 \in A(X_3, C_3)$, both X_1 and X_2 must be in S. Therefore, the deadlock configuration must include a C_2 packet in X_1 and a C_1 packet in X_2. However, it is impossible to simultaneously route a C_2 packet to X_1 and a C_1 packet to X_2, so the deadlock configuration is not routable, which is a contradiction. □

Lemma 3.2 *There is no ordering of the queues shown in Figure 1 such that every packet always has the possibility of moving to a higher ordered queue.*

Proof: Assume for the sake of contradiction that such an ordering is possible. Because $A(X_1, C_2) = \{X_3\}$ and $A(X_2, C_1) = \{X_3\}$, queue X_3 must be higher ordered than both X_1 and X_2. However, $A(X_3, C_3) = \{X_1, X_2\}$, so either X_1 or X_2 (or both) must be higher ordered than X_3, which is a contradiction. □

Combining the two previous lemmas yields the following theorem.

Theorem 3.3 *There exists an adaptive routing algorithm which is free of deadlock, and for which there is no ordering of the queues such that every packet always has the possibility of moving to a higher ordered queue.*

4 Restrictions of Adaptive Routing Algorithms

In this section we will show that every deadlock-free, adaptive packet routing algorithm can be restricted to obtain an oblivious algorithm which is also deadlock-free. The proof will depend on the following lemma.

Lemma 4.1 *Let A be any deadlock-free, adaptive routing algorithm, let q_1 be any queue, and let c_1 be any color such that $|A(q_1, c_1)| \geq 2$. Let q_2 be any queue such that $q_2 \in A(q_1, c_1)$, and let B be the restriction of A obtained by removing q_2 from the waiting set associated with (q_1, c_1). If B is subject to deadlock, then there must exist some routable deadlock configuration for B in which queue q_1 contains a packet with color c_1.*

Proof: Because B is subject to deadlock, there must exist some configuration T which is a deadlock configuration for B and which is routable by B. Because B is a restriction of A, it follows that configuration T is also routable by A. However, A is deadlock-free, so T must not be a deadlock configuration for A. Therefore, queue q_1 must have color c_1 in configuration T. □

Theorem 4.2 *Given any adaptive, deadlock-free routing algorithm A, there exists an oblivious, deadlock-free routing algorithm B which is a restriction of A.*

Proof Sketch: Assume for the sake of contradiction that the claim is false. Then there must exist some adaptive, deadlock-free routing algorithm A such that every routing algorithm A' which is a restriction of A is subject to deadlock. Let A be such a deadlock-free routing algorithm, let q_1 be any queue, and let c_1 be any color such that $|A(q_1, c_1)| \geq 2$. Let q_2 and q_3 be any distinct queues such that $q_2 \in A(q_1, c_1)$ and $q_3 \in A(q_1, c_1)$, let A' be the restriction of A obtained by removing q_2

from the waiting set associated with (q_1, c_1), and let A'' be the restriction of A obtained by removing q_3 from the waiting set associated with (q_1, c_1). It follows from Lemma 4.1 that there exists a configuration T' (similarly, T'') which is a routable deadlock configuration for A' (similarly, A'') and in which queue q_1 contains a packet with color c_1. Let $T = T' \oplus T''$ and let S be the critical set of T. Let R be the configuration which also has critical set S but in which all of the queues are empty. Note that the following properties hold:

Property 1: Configuration T is a deadlock configuration for A.

Property 2: Configuration R is a routable configuration for A.

Property 3: The set S is the critical set of both configuration T and configuration R.

Property 4: Every nonempty queue q in R has a color c such that $A(q, c) \subseteq S$.

We will define an algorithm for transforming R and T while maintaining Properties 1 through 4 listed above. The algorithm will repeatedly add packets to empty queues in R until none of the queues in R is empty. At this point R will be a routable deadlock configuration, which will be the desired contradiction. The algorithm for transforming R and T consists of repeatedly performing the following subroutine until R contains no empty queues.

First, select an arbitrary queue q which is empty in R. Let $c = T(q)$. Because queue q is reachable by some packet p with color c (from the definition of a configuration), there must exist a simple path from p's injection queue to queue q. Define the configuration P in which the critical set consists of all of those queues that appear in this simple path, and in which all of the queues in the critical set contain a packet with color c. Transform R to become the configuration obtained by adding P and R (that is, perform the assignment $R \leftarrow P \oplus R$), transform T to become the configuration obtained by adding P and T (that is, perform the assignment $T \leftarrow P \oplus T$), and let S be the critical set of the transformed configurations R and T. Note that at this point R is routable, because it is possible to first route packets with the desired colors to all of the nonempty queues in R which are not in P and then fill the queues in P with packets with color c. Also, note that at this point T may not be a deadlock configuration, because it is possible that some of the packets in P have waiting sets that include queues which are not in S.

Next, select an arbitrary queue q' in the simple path described above such that $A(q', c) \not\subseteq S$ (if such a queue exists). Let q'' be the successor of q' in the simple path described above (note that q'' must exist if q' exists, because $A(q, c) \subseteq S$ so $q' \neq q$). Let A' be the restriction of A obtained by removing q'' from the waiting set associated with (q', c). It follows from Lemma 4.1 that there exists a configuration D which is a routable deadlock configuration for A' and in which queue q' contains a packet with color c. Let D' be the configuration with the same critical set as D but in which all of the queues are empty. Transform R to become the configuration obtained by adding R and D' (that is, perform the assignment $R \leftarrow R \oplus D'$), transform T to become the configuration obtained by adding T and D (that is, perform the assignment $T \leftarrow T \oplus D$), and let S be the critical set of the transformed configurations R and T. Repeat this procedure of selecting a queue q' in the simple path such that $A(q', c) \not\subseteq S$ and transforming R, T and S until no such queue q' exists. When no such queue q' exists, return from the subroutine.

Note that upon returning from the subroutine Properties 1 through 4 above must hold. Also, note that any queue in R which was nonempty before calling the subroutine will again be nonempty after calling the subroutine. Finally, note that following the call to the subroutine, R contains at least one additional nonempty queue. Because the number of queues is finite, this procedure must terminate, at which point R is both routable by A and a deadlock configuration for A, which is a contradiction. □

5 Minimal Routing in Cycle and Torus Networks

In this section we will prove lower bounds on the number of queues per node that are required for deadlock-free, minimal routing in cycle and torus networks. Our approach will be to first prove a lower bound on the queue requirements of deadlock-free, minimal, oblivious routing algorithms for cycle networks. We will then use this lower bound, along with Theorem 4.2 and the fact that a torus network can be decomposed into disjoint cycles, to obtain a lower bound on the queue requirements of deadlock-free, minimal routing algorithms for both cycle and torus networks.

Lemma 5.1 *Let routing algorithm A be any deadlock-free, minimal, oblivious routing algorithm for a cycle network with n nodes. The cycle network must contain at least $3n - 12$ standard queues.*

Proof: Because A is deadlock-free and oblivious, it follows that there exists an ordering of the queues such that every packet visits the queues in ascending order [18]. Let $k = \lfloor n/2 \rfloor - 1$ (so either $n = 2k + 2$ or $n = 2k + 3$). We will say that a packet is routed in the *clockwise* direction if it visits queues in nodes of the form $i, (i + 1) \bmod n, (i + 2) \bmod n, \ldots, j$, and in the *counterclockwise* direction otherwise. Note that for each node i, $0 \leq i < n$, algorithm A routes packets from node i to node $(i + k) \bmod n$ in the clockwise direction. Therefore, for each node i, $0 \leq i < n$, there must exist an ascending sequence of standard queues $s_{i,i}, s_{i,(i+1) \bmod n}, \cdots, s_{i,(i+k) \bmod n}$ where each queue of the form $s_{i,j}$ is located in node j (for example, let $s_{i,j}$ be the highest ordered standard queue in node j which is visited by a packet with source node i and destination node $(i + k) \bmod n$). For each i, $0 \leq i < n$, let $S_i = s_{i,i}, s_{i,(i+1) \bmod n}, \cdots, s_{i,(i+k) \bmod n}$ denote the ascending sequence of standard queues beginning in node i. Let $h = n - 1 - k$, note that $S_h = s_{h,h}, s_{h,h+1}, \ldots, s_{h,n-1}$, and note that S_0 and S_h are disjoint.

We will say that a sequence of queues is a *clockwise-increasing* (similarly, *counterclockwise-increasing*) sequence if when the queues are visited in ascending order, the nodes containing the queues are visited in clockwise (counterclockwise) order. We will show that there must exist at most three mutually disjoint clockwise-increasing sequences of queues, the total length of which is at least $n + k$. There are two cases:

Case 1: There exists a clockwise-increasing sequence of standard queues $X = x_0, x_1, \ldots, x_{n-1}$ such that for each i, $0 \leq i < n$, x_i is located in node i. In this case, we have two subcases:

Case 1a: There exists an a, $0 \leq a \leq n - 1$, such that S_a and X are disjoint. In this case, the two disjoint clockwise-increasing sequences are S_a and X, and their total length is $n + k + 1$.

Case 1b: For each i, $0 \leq i \leq n - 1$, S_i and X intersect. In this case, let a be the largest value of i, $0 \leq i \leq n-1$, such that there exists a value $i' \geq i$ where $s_{i,i'} = x_{i'}$. Let a' be any value such that $a' \geq a$ and $s_{a,a'} = x_{a'}$. Let $b = (a+1) \bmod n$ and let b' be any value such that $s_{b,b'} = x_{b'}$. Note that $a' \geq a \geq k \geq b'$. Let Y be the sequence

$$s_{b,b}, \cdots, s_{b,b'-1}, x_{b'}, \cdots, x_{a'},$$

$$s_{a,a'+1}, \cdots, s_{a,(a+k) \bmod n}.$$

The sequence Y is clockwise-increasing and has length $n + k$.

Case 2: There does not exist such a sequence X. In this case, we have two subcases:

Case 2a: There exists an a, $0 \leq a \leq h$, such that S_a and S_0 are disjoint and such that S_a and S_h are disjoint. In this case, the three disjoint clockwise-increasing sequences are S_0, S_a, and S_h, and their total length is $3k + 3 \geq n + k$.

Case 2b: For each i, $0 \leq i \leq h$, either S_i and S_0 intersect or S_i and S_h intersect, but not both. In this case, let a be the largest value

of i in the range $0 \leq i \leq h$ such that S_i and S_0 intersect. Let a' be any value such that $s_{a,a'} = s_{0,a'}$. Let $b = a + 1$ and let b' be any value such that $s_{b,b'} = s_{h,b'}$ (note that such a b' must exist because of the definition of a and the fact that S_h does not intersect S_0). Let Y be the sequence

$$s_{0,0}, \ldots, s_{0,a'-1}, s_{a,a'}, \ldots, s_{a,a+k}$$

and let Z be the sequence

$$s_{b,b}, \ldots, s_{b,b'}, s_{h,b'+1}, \ldots, s_{h,n-1}.$$

Note that Y and Z are clockwise-increasing sequences and that they must be disjoint (because otherwise there would exist a clockwise-increasing sequence X spanning all of the nodes). Also, note that the length of Y is $a+k+1$ and the length of Z is $n-a-1$, so their total length is $n+k$.

Thus, in any case there must exist at most three mutually disjoint clockwise-increasing sequences of queues, the total length of which is at least $n+k$. An analogous argument can be used to show that there must exist at most three mutually disjoint counterclockwise-increasing sequences of queues, the total length of which is at least $n+k$. Because a clockwise-increasing sequence of queues and a counterclockwise-increasing sequence of queues can intersect in at most one queue, it follows that the entire collection of clockwise-increasing sequences and counterclockwise-increasing sequences contains at least $(n+k)+(n+k)-(3*3) = 2n+2k-9 \geq 3n-12$ distinct queues. □

Theorem 5.2

Let routing algorithm A be any deadlock-free, minimal routing algorithm for a cycle network with 13 *or more nodes or for a torus network in which at least one of the dimensions is of length* 13 *or greater. The cycle or torus network must contain at least one node which has three or more standard queues.*

Proof: The claim for a cycle network follows immediately from Theorem 4.2 and Lemma 5.1. The claim for a torus network follows from Theorem 4.2, Lemma 5.1, and the observation that a torus can be decomposed into disjoint cycles such that all minimal length paths between pairs of nodes within a cycle lie within the cycle. □

References

[1] Baruch Awerbuch, Shay Kutten, and David Peleg. Efficient deadlock-free routing. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 177–188, 1991.

[2] Robert Cypher and Luis Gravano. Adaptive, deadlock-free packet routing in torus networks with minimal storage. Technical Report RJ 8571, IBM Almaden Research Center, January 1992. Also to appear in *Proc. 1992 Intl. Conf. on Parallel Processing.*

[3] W. J. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36:547–553, May 1987.

[4] Sergio A. Felperin, Hernán Laffitte, Guillermo Buranits, and Jorge L.C. Sanz. Deadlock-free minimal packet routing in the torus network. Technical Report TR:91-22, IBM Argentina, CRAAG, 1991.

[5] B. Gavish, P.M. Merlin, and P.J. Schweitzer. Minimal buffer requirements for avoiding store-and-forward deadlock. Technical Report RC 6672, IBM T.J. Watson Research Center, August 1977.

[6] Inder S. Gopal. Prevention of store-and-forward deadlock in computer networks. *IEEE Transactions on Communications*, 33(12):1258–1264, December 1985.

[7] Luis Gravano, Gustavo D. Pifarré, Sergio A. Felperin, and Jorge L.C. Sanz. Adaptive deadlock-free worm-hole routing with all minimal paths. Technical Report TR:91-21, IBM Argentina, CRAAG, 1991.

[8] K.D. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, 29(4), April 1981.

[9] Peter A.J. Hilbers and Johan J. Lukkien. Deadlock-free message routing in multicomputer networks. *Distributed Computing*, 3:178–186, 1989.

[10] C.R. Jesshope, P.R. Miller, and J.T. Yantchev. High performance communications in processor networks. In *Proc. 16th Intl. Symposium on Computer Architecture*, pages 150–157, 1989.

[11] P. Kermani and L. Kleinrock. Virtual Cut-Through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[12] S. Konstantinidou. Adaptive, minimal routing in hypercubes. In *Proc. 6th. MIT Conference on Advanced Research in VLSI*, pages 139–153, 1990.

[13] S. Konstantinidou and L. Snyder. The Chaos router: A practical application of randomization in network routing. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 21–30, 1990.

[14] Yishay Mansour and Boaz Patt-Shamir. Greedy packet scheduling on shortest paths. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 165–175, 1991.

[15] F.M. Merlin and P.J. Schweitzer. Deadlock avoidance in store-and-forward networks. 1: Store-and-forward deadlock. *IEEE Transactions on Communications*, 28(3):345–354, March 1980.

[16] Yoram Ofek and Moti Young. Principles for high speed network control: loss-less and deadlock-freeness, self-routing and a single buffer per link. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 161–175, 1990.

[17] Gustavo D. Pifarré, Luis Gravano, Sergio A. Felperin, and Jorge L.C. Sanz. Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 278–290, 1991.

[18] Sam Toueg and Kenneth Steiglitz. Some complexity results in the design of deadlock-free packet switching networks. *SIAM Journal on Computing*, 10(4):702–712, November 1981.

[19] Sam Toueg and Jeffrey D. Ullman. Deadlock-free packet switching networks. *SIAM Journal on Computing*, 10(3):594–611, August 1981.

[20] J. Yantchev and C.R. Jesshope. Adaptive, low latency, deadlock-free packet routing for networks of processors. *IEE Proc., Pt. E*, 136(3):178–186, May 1989.

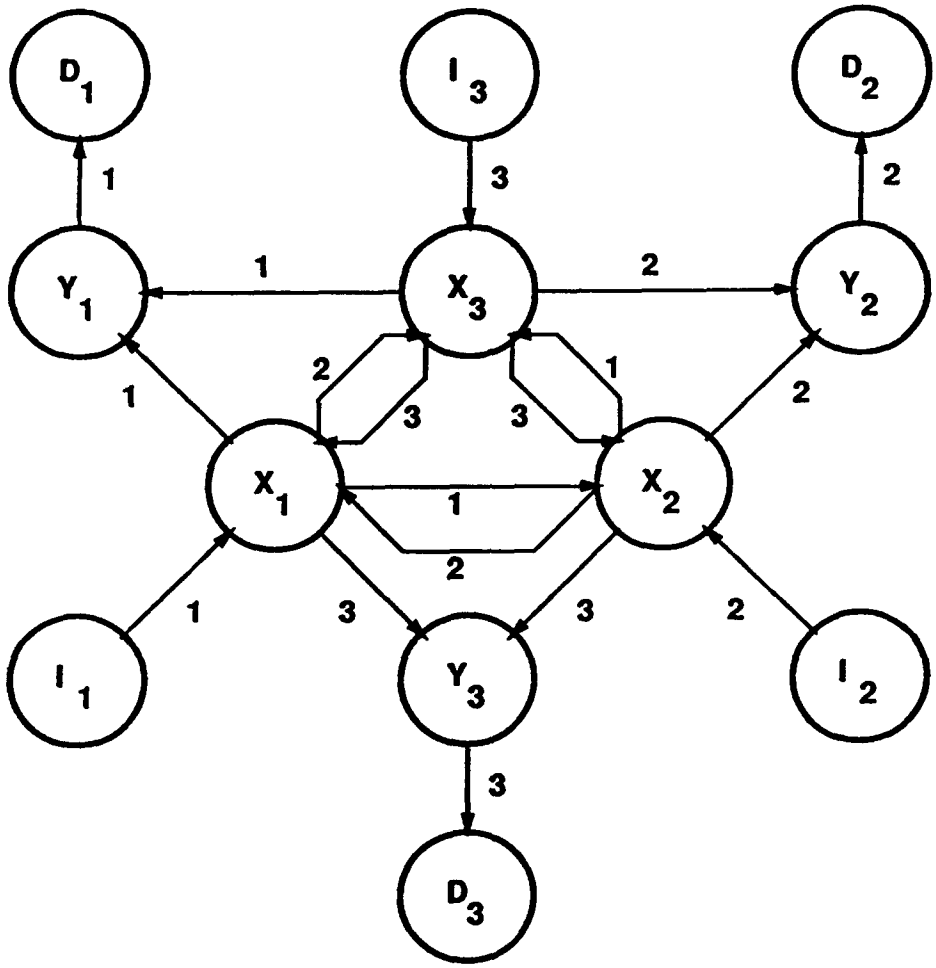


Figure 1: A deadlock-free, adaptive routing algorithm A for which the technique of ordering the queues cannot be used to prove freedom from deadlock.

The Slide Mechanism with Applications in Dynamic Networks

(Extended Abstract)

Yehuda Afek* Eli Gafni† Adi Rosén‡

Abstract

This paper presents a simple and efficient building block, called *slide*, for constructing communication protocols in dynamic networks whose topology frequently changes. We employ *slide* to derive (1) an end-to-end communication protocol with optimal *amortized* message complexity, and (2) a general method to efficiently and systematically combine dynamic and static algorithms. (Dynamic algorithms are designed for dynamic networks, and static algorithms work in networks with stable topology.)

The new end-to-end communication protocol has amortized message communication complexity $O(n)$ (assuming that the sender is allowed to gather enough data items before transmitting them to the receiver), where n is the total number of nodes in the network (the previous best bound was $O(m)$, where m is the total number of links in the network). This protocol also has bit communication complexity $O(nD)$, where D is the data item size in bits (assuming data items are large enough; i.e., for $D = \Omega(nm \log n)$). In addition we give, as a byproduct, an end-to-end communication protocol using $O(n^2m)$ messages per data item, which is considerably simpler than other protocols known to us (the best known end-to-end protocol has message complexity $O(nm)$[AG91]). The protocols above combine in an interesting way several ideas: the *information dispersal algorithm* of Rabin [Rab89], the *majority* insight of [AFWZ88, AAF+], and the *slide* protocol.

The second application of *slide* develops a systematic mechanism to combine a dynamic algorithm with a static algorithm for the same problem, such that the combined algorithm automatically adjusts its communication complexity to the network conditions. That is, the combined algorithm solves the problem in a dynamic network, and if the network stabilizes for a long enough period of time then the algorithm's communication complexity matches that of the static algorithm. This approach has been first introduced in [AM88] in the context of topology update algorithms.

1 Introduction

One of the most important tasks of distributed algorithm designers is to construct *simple* and efficient building blocks for various network models. While simple and efficient building blocks for synchronous and asynchronous static networks have been presented [Awe85, AAG87, CL85, BGP89, AP90, DS80], only complicated, though theoretically efficient, algorithms are known for dynamic networks whose topology frequently change (like the ARPANET ([MRR80]) and DECNET ([Wec80])).

In this paper we present a simple and efficient building block, called *slide*, for dynamic networks with frequently changing topologies (i.e. for the ∞-delay model [AG88]). Essentially, *slide* establishes a non-fifo virtual communication link between two arbitrary nodes in the network. Its effectiveness is demonstrated in three applications:

- An end-to-end protocol that is considerably simpler then any previous known protocol and whose message communication complexity is $O(n^2m)$ (compared to the best known $O(nm)$ messages),
- An $O(n)$ *amortized* message complexity end-to-end protocol, assuming that the sender is allowed to gather enough data items before transmitting them to the receiver, and
- A mechanism that senses topological changes and controls the amortized (per topological change)

*Computer Science Department, Tel-Aviv University, Tel-Aviv 69978, Israel, and AT&T Bell Laboratories.

†Computer Science Department, U.C.L.A., Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

‡Computer Science Department, Tel-Aviv University, Tel-Aviv 69978, Israel.

PoDC '92-8/92/B.C.

© 1992 ACM 0-89791-496-1/92/0008/0035...$1.50

communication complexity of any dynamic algorithm. If a static algorithm is run in parallel with a dynamic algorithm controlled by the mechanism the resulting algorithm adjusts its communication complexity to the network conditions.

In addition to these applications, the *slide* protocol has already proved its versatility and applicability in [APSV91] where it was shown that both it and its first two applications above (with minor changes), are self-stabilizing (as a result the third application also preserves the self-stabilization property).

The end-to-end protocols presented in this paper are data non-oblivious. The transport protocol uses the data. The first application, the *majority algorithm*, employs the majority idea introduced in [AFWZ88, AAF+], therefore using the contents of the delivered data-items to make its decisions. Our second application, the *data dispersal algorithm*, breaks each large data item into many packets (using the information dispersal algorithm of Rabin [Rab89]) and achieves an $O(n)$ communication complexity (assuming large data items, $\Omega(nm \log n)$ bits). This is in contrast to other known end-to-end communication algorithms, which are all data-oblivious, and require that each data item as a whole traverses each link in the network, resulting in $O(m)$ communication complexity. Our solution does not yield $O(n)$ communication complexity algorithms for smaller data items, since it breaks each data item into $O(nm)$ pieces.

These algorithms designed to tolerate faults have high communication complexity. They incur this complexity even when the network is stable (the obvious end-to-end communication algorithm for stable networks has $O(n)$ communication complexity). To adapt the complexity to the case at hand the second part of this paper uses *slide* to develop a systematic method to alleviate this drawback by combining a dynamic algorithm with a static one to get a new dynamic algorithm whose communication complexity matches that of the static algorithm if the network becomes static, and still works correctly even if the network topology never stabilizes. The method applies to on-line algorithms in the sense that these algorithms receive input values during their run, they never terminate and they produce a series of output values (like end-to-end communication algorithms). The complexity of such algorithms in dynamic networks is usually measured as the maximum cost they incur between any two successive output events. In order to capture in one complexity measure the complexity of the combined algorithm both when the network is dynamic and when topological changes cease, we give a two component amortized cost: The cost per topological change plus the cost per output (input) event. The former captures the cost of the algorithm when there are topological changes in the network, while the latter captures the cost in times when the network topology is stable. Given a static algorithm with communication complexity C_s and a dynamic algorithm, we construct a combined algorithm with amortized communication complexity of $O(n + C_s + m)$ per topological change plus $O(C_s)$ per output event.

The implementation of this method depends on the possibility to merge the outputs of two algorithms that in parallel solve the same problem, into one output stream. The merging itself is not systematic and depends on the specific problem being solved. Such a merging mechanism for the end-to-end problem was given in [AG88] and is used here to exemplify the method, resulting in an algorithm that runs in dynamic networks, and achieves $O(n)$ communication complexity if the network stabilizes.

1.1 Related work

Techniques The algorithms in this paper combine several techniques from recently reported research. The *slide* mechanism combines in an interesting way ideas from [AMS89] with a technique appearing in [MS80]. Combining *slide* with a concept introduced in [AFWZ88, AAF+], we construct an end-to-end algorithm, the *majority* algorithm. This algorithm, in conjunction with the information dispersal algorithm (IDA) of Rabin [Rab89], yields our data dispersal algorithm.

In the second part of the paper we combine the *slide* protocol with a an idea introduced in [AM88], to generate a simple and systematic method for combining a dynamic version of an algorithm with a static version. The method is exemplified on the end-to-end problem by combining the bootstrap mechanism for dynamic networks of [AG91] with a static broadcast and echo along a path, to create a dynamic algorithm, that automatically adjusts its communication complexity to the network conditions.

The end-to-end problem One approach to solve the end-to-end problem in a dynamic network is to deliver the data items over a fixed path and to construct a new path every time a link on the path fails. Implementations of this approach appear in [Fin79, AAG87, AS88, AAM89]. This approach requires, however, the whole network to stabilize for a period of time allowing the construction of the path and the communication over it [AAG87, AS88, AAM89], or at least requires links forming some path between the sender and the receiver to be operational for such a period of time [AGH90].

In [AE86] it is stated that the *eventual connectivity* fairness condition, namely that there is no edge-cut all of whose links are permanently down, is sufficient for communication between the sender and the receiver. The problem is solved in [AE86, Vis83] under this condition

by using *sequence numbers* to number the data items sent by the sender. This approach yields theoretically **unbounded** algorithms, as the sequence numbers grow with the number of data items transmitted.

In recent years a sequence of works gave bounded, and increasingly efficient, solutions to the problem. The first **bounded** end-to-end communication algorithm under the *eventual connectivity* fairness condition was presented in [AG88]; however, this solution has exponential message complexity per data item. Subsequently, in [AMS89], the first **polynomial** message complexity algorithm was presented, having bit communication complexity of $O(n^9 + mD)$. Recently, a new resynchronization protocol for dynamic networks was developed in [AG91] and was used to construct an end-to-end communication algorithm whose bit communication complexity is $O(nm \log n + mD)$.

Outline: We start by introducing the model in Section 2 and give a description of the *slide* and the end-to-end communication protocols in Section 3; An outline of proof of correctness and complexity analysis is given in the appendix. Section 4 contains the method to turn any dynamic algorithm into a dynamic algorithm that adjusts its communication complexity to network conditions.

2 The Model

We consider a communication network in the form of a graph $G(V, E)$, $|V| = n$, $|E| = m$, where the nodes are processors and the edges are undirected communication links. Each *undirected* link consists of two *directed* links, delivering messages in opposing directions.

Each link has *bounded* capacity. By capacity we refer to the maximal number of messages in transit on a given link at a given time. The bounded capacity can be either constant, or a function of the network size. For clarity of description, we consider networks in which each link has $O(n)$ capacity. As noted in the analysis our algorithms can be easily modified to conform to the constant capacity model without effecting their complexities.

The communication over the links obey the FIFO discipline, and no bound on the transmission delay time is known. A directed link is *non-viable* if starting from some message it does not deliver any message. The transmission delay time for this message and the subsequent messages sent on this link is considered infinite. A directed link is *viable* if any message sent over it is eventually delivered. Similarly, an undirected link is *viable* if both of its directed links are viable. (We assume that link that fails, fails in both directions.) Two nodes in the network are *eventually connected* if there is a path of viable links connecting them.

The model defined here is called the "∞-delay model" in [AG88], and as stated there the model of failing and recovering links [AAG87] can easily be reduced to it. When combining dynamic algorithms with static algorithms (Section 4) we consider the model of [AAG87] in which topological changes are part of the input to the algorithm. The reduction of the changing topology model of [AAG87] to the ∞-delay model, adds at most $O(1)$ messages per topological change to the protocol's communication complexity in the ∞-delay model.

2.1 The End to End Communication Problem

An end-to-end communication algorithm transmits a sequence of data items from a sender to a receiver. The data items are input to the sender on-line, that is the sequence is not known at the beginning of the operation of the algorithm and the data items are input one by one.

The algorithm must satisfy the following properties:

- **Safety:** At any time the sequence of the data items output by the receiver is a prefix of the sequence of data items input by the sender.
- **Liveness:** If the sender and the receiver are eventually connected, then every data item input by the sender will eventually be output by the receiver.

The Complexity Measures Since the algorithm transmits a sequence of data items input one by one, the complexity analysis should evaluate the performance of the algorithm per data item. This evaluation is done by measuring the cost of the algorithm between any two successive data item output events at the receiver.

We consider the following complexity measures: *Message Complexity:* The number of messages transmitted in the network in the worst case between any two successive output events at the receiver; *Bit Communication Complexity:* The number of bits transmitted in the network in the worst case between any two successive output events at the receiver; *Space Complexity:* The maximal amount of space required at each node, per incident link, measured in bits of memory.

3 Description of the Algorithms

When presenting the code of the algorithms we use the *guarded commands* language of Dijkstra [DF88], where the code of each process is in the form $G_1 \rightarrow A_1 \Box G_2 \rightarrow A_2 \Box \ldots G_l \rightarrow A_l \Box$. The code is executed by repeatedly selecting an arbitrary i from all guards G_i which are true and executing A_i. A guard G_i is a conjunction of predicates. The predicate **Receive** M is true when a

message M is available to be received. If the statements associated with this predicate are executed, then prior to this execution the message M is received. The message may contain some values that are assigned, upon its receipt, to variables stated in the **Receive** predicate (e.g. **Receive** TOKEN(data)).

The following sections describe the various algorithms; An outline of the correctness proofs and analysis is added in the appendix.

3.1 The *slide* Protocol

In this protocol one node, the sender, inputs tokens into the network, and one node, the receiver, outputs the tokens. The protocol guarantees that tokens are neither lost nor duplicated, and that the total number of tokens in the network at any given time is bounded. If the sender and the receiver are eventually connected, then eventually the input of a new token to the network is enabled at the sender. However, tokens are output at the receiver not necessarily in the order they are input at the sender. Thus the *slide* establishes between the sender and the receiver a non-fifo, bounded capacity virtual communication link that neither loses nor duplicates messages.

The *slide* protocol is implemented by storing and transferring tokens between the nodes of the network as follows: Each undirected link is considered as two antiparallel links. Each node maintains for each incident incoming link a pile of slots numbered 1 to n. Each slot has room for one token, and each pile is used to store tokens arriving on the link associated with it; Tokens from a pile can be forwarded over any outgoing link. The crux of the protocol is that a token is sent from any slot i at node v to slot j at the (v,u) pile at node u, only if $j < i$. To this end the nodes maintain for each outgoing link a variable, called *bound*, holding an upper bound to the lowest numbered slot available at the other side of the link. The tokens are sent from slots with a number higher than the bound, and thus are guaranteed to conform to the above rule. Every time a token is removed from a pile, a signal to this effect is sent over the incoming link associated with the pile. Since the only source of tokens for a specific pile is the node on the other end of its associated link, the bound can be maintained by incrementing it every time a token is sent over the link, and decrementing it every time a signal is received over the link. Thus the bound is never smaller than the number of tokens in the pile on the other side of the link plus the number of tokens in transit over the link.

New tokens enter the network only at the sender, to a special slot at level n, and only when this slot is vacant. The receiver has always a vacant slot of level 1, and removes and outputs any token it receives. If the sender and the receiver are eventually connected, then

```
Initialization ⟶
    for every incident link e
            bound[e]:=1;
            top[e]:=0;
□
Receive SIGNAL on e ⟶
    bound[e]:=bound[e]-1;
□
Receive TOKEN(data) on e ⟶
    top[e]:=top[e]+1;
    slots[e][top[e]]:=data;
□
∃e, e' s.t. top[e'] > bound[e] ⟶
            /* e' not necessarily ≠ e */
    send TOKEN(slots[e'][top[e']]) on e;
    send SIGNAL on e';
    top[e']:=top[e']-1;
    bound[e]:=bound[e]+1;
□
```

a: ordinary's node code

```
Initialization ⟶
    input_pile[n]:=vacant;
□
input_pile[n]=vacant ⟶
    input_pile[n]:=next input;
□
input_pile[n]≠vacant
            and ∃e s.t. bound[e]< n ⟶
    send TOKEN(input_pile[n]) on e;
    input_pile[n]:=vacant;
    bound[e]:=bound[e]+1;
□
```

b: additions for the sender

```
Receive TOKEN(data) on e ⟶
    output(data);
    send SIGNAL on e ;
□
```

c: receiver's code

Figure 1: The *slide*

eventually the special slot at the sender is vacant. Thus the tokens slide in the network from the sender to the receiver by sliding from higher numbered slots to lower numbered slots as they travel over links. Clearly, each token can make at most n hops in the network. Since the protocol maintains for each link $2n$ slots, and (as we prove in the sequel) at most $2n$ tokens can be in transit on each link at any given time, the total number of tokens in the network is at most $O(nm)$. This is the capacity of the *slide*, denoted *Cap*.

The code of the algorithm is given in Figure 1; It uses two types of messages: TOKEN messages which are used to transfer the tokens themselves, and SIGNAL messages that are used to inform over a link that a token from the pile associated with it was removed from the pile. The differences between the sender and an ordinary node are due to the fact that the sender is the node that inputs new tokens to the network. The code of the sender is the code of an ordinary node and in addition the code appearing in figure 1b. Note that the sender is disabled to input new tokens if input_pile[n] is not vacant. The receiver does not send tokens and outputs any token it receives. Therefore, its code is restricted to the code appearing in figure 1c.

In the following subsections we present two solutions to the problem of end-to-end communication in dynamic networks, that use the *slide* as a building block. The second solution, the *data dispersal* algorithm, is our main result that achieves linear communication cost for large enough data items.

3.2 The Majority Algorithm

Given the *slide* we construct a simple end-to-end communication algorithm by operating the *slide* from the sender (S) to the receiver (R) and combining it with an idea of [AFWZ88, AAF+]. Whenever S wishes to send a data-item to R it sends consecutively $2 \cdot Cap + 1$ duplicates of the data item to R using the *slide*. To receive the first data item R waits for $Cap + 1$ data items and outputs one of them. For each subsequent data item it waits for $2 \cdot Cap + 1$ data items, takes the majority of the values received, and outputs this value. Since no more than *Cap* data items can be delayed in the network at any given time, the majority of the data items received represent the next data item (see Theorem 18 in the appendix).

The sender's algorithm, described below in Figure 2, interfaces with the sender's algorithm of the *slide* protocol in a way that each token *sent* here is input by the *slide* sender, and each token output by the *slide* receiver is *received* by the receiver of the majority algorithm.

The bit communication complexity of this algorithm is $O(n^2 mD)$, where D is the size of the data item.

We remark that by adding a toggle bit to each data item the protocol can be made robust against leftover

```
Initialize ⟶
    items-set:= φ ;
    first_item:=true ;
□
Receive(data-item) ⟶
    items-set: = items-set ∪ data-item ;
    call check_and_output ;
□
```

a: receiver's code

```
true ⟶
    data-item:=next input;
    for i:=1 to 2·Cap + 1 do
                Send(data-item);
    od
□
```

b: sender's code

```
Procedure check_and_output
    if first_item=true and |items-set| = Cap + 1 then
                    /* first data item */
        output(any_data_item_of(items-set)) ;
        items_set := φ ;
        first_item:=false ;
    else if first_item=false and |items-set| = 2·Cap + 1
        then        /* all other data items */
            output(majority(items-set)) ;
            items_set := φ ;
        endif
    endif
```

c: procedure check_and_output

Figure 2: The Majority Algorithm

data items that can be in the network before the algorithm has started.

3.3 The Data Dispersal Algorithm

For the cases were the data items are large with respect to the size of the network, i.e. having size of $\Omega(nm \log n)$ bits, we construct an algorithm that achieves linear ($O(n)$) bit communication complexity. The same algorithm can also be used for smaller data items if the sender is allowed to wait for more than one data item and transmit several data items together.

In order to derive this algorithm, we combine Rabin's Information Dispersal Algorithm [Rab89] with the fact that the *slide* can delay only a bounded number of packets. Using the Information Dispersal Algorithm (IDA), the sender splits the data item into packets and sends them to the receiver over the *slide*. Since the IDA allows the construction of the data item from a subset of the packets, we can tolerate the loss of the bounded number of packets that the *slide* can hold.

The sender creates using the IDA $2 \cdot Cap + 1$ packets, each of size $O(\frac{D}{Cap+1})$ bits, where D is the size of the data item; The receiver is thus able to construct the full data item from only $Cap + 1$ packets. The sender sends the $2 \cdot Cap + 1$ packets, and as the *slide* can delay at most Cap of them the receiver will receive enough packets to reconstruct the data item. The only problem left is to make sure that the receiver does not use old delayed packets to reconstruct subsequently sent data items. To alleviate this problem the sender adds to the packets of each data item a label. The receiver outputs the first data item after calculating it from the first $Cap + 1$ packets it receives; For each subsequent data item it waits for $2 \cdot Cap + 1$ packets, checks which label has the majority among the labels in the packets, and uses only the packets having this label. For each new data item the sender must use a label that is not present in the network. Therefore the receiver sends back to the sender every packet it receives through another *slide* operated in the opposite direction. Thus the sender always knows which labels are present in the network. As the capacity of each *slide* is bounded by Cap, $2Cap+1$ different labels suffice.

In the code, presented in Figure 3, we use the subscripts $*_{S \to R}$ and $*_{R \to S}$ to denote operations with respect to the *slide* from the sender to the receiver and the *slide* from the receiver to the sender, respectively. The interface with the *slide* protocols is similar to this interface for the majority algorithm. $\mathcal{L}$ denotes a set of $2 \cdot Cap + 1$ labels. Note that the function *extract* can extract an arbitrary member from the set it is applied to.

The bit communication complexity of the data dispersal algorithm is $O(nD)$, if it is applied to data items of size $\Omega(nm \log n)$ bits. If the algorithm is applied to smaller data items, it achieves an amortized bit communication complexity of $O(nD)$, by combining several data items together.

4 Combining Static and Dynamic Algorithms

Dynamic algorithms that are designed to operate correctly in eventually connected dynamic networks usually suffer the drawback that even if the network becomes static their communication complexity does not decrease. In this section we use a variation of the *slide* protocol, called *generalized slide* to present a systematic methodology to combine a static algorithm with a dynamic one into a single dynamic algorithm whose communication complexity matches that of the static algorithm if the network topology becomes static for a large enough period of time, and operates correctly even if the network topology changes frequently. The approach was first introduced in [AM88] in an ad-hoc manner for a specific problem, the topology update problem.

The essence of our method is a mechanism ensuring that the dynamic algorithm sends no more than k messages per topological change (i.e. its amortized communication complexity [AAG87, AM88] is $O(k)$ per topological change), where k is a design parameter. In particular, if topological changes cease, the dynamic algorithm grindes to a halt. To ensure that progress is made in that case (if the network topology stabilizes) we run in parallel a static algorithm used in conjunction with the reset procedure of [AAG87]. Since both algorithms run in parallel, their output should be merged into a single series of output events for the problem being considered. The implementation of the merging mechanism depends on the particular problem; As an example we describe below how to merge two algorithms in the context of the end-to-end communication problem.

4.1 Generalized *slide*

The *slide* is generalized to a distributed object which interacts with two sets of nodes, the *producers* and the *consumers*. Producers deposit tokens into the *slide* and consumers may consume tokens, thus extracting them from the *slide* locally. The generalized *slide* has the following properties: (1) At any time, the total number of tokens consumed is no more than the total number of tokens produced, (2) the total number of tokens stored in the *slide* at any given time, called the *capacity*, is bounded, (3) each token performs at most n hops in the network, and (4) if a *consumer* wishes to consume tokens and is eventually connected to a *producer* that infinitely many times produces tokens, then eventually the consumer will have a token to consume. We achieve

```
Initialization ⟶
    $\overline{\mathcal{L}} := \mathcal{L}$ ;
    sending:=false;
    missing:= 0;
□
sending=false and missing ≤ 2·Cap ⟶
    data-item:=next input;
    l:=extract($\overline{\mathcal{L}}$);
    using the IDA with parameters
          Cap + 1 and 2·Cap + 1,
          create packets number 1 to 2·Cap + 1;
    send_buffer:=$\bigcup_{i=1}^{2Cap+1}$ (l,i,packet$_i$);
    count[l]:=0;
    sending:=true;
□
sending=true ⟶
    (l,i,packet):=extract(send_buffer);
    Send$_{S\to R}$ (l,i,packet);
    count[l]:=count[l]+1;
    missing:=missing+1;
    if |send_buffer|=0 then sending:=false ; endif
□
Receive$_{R\to S}$ (l,i,packet) ⟶
    missing:=missing-1;
    count[l]:=count[l]-1;
    if (count[l]=0) then $\overline{\mathcal{L}}$= $\overline{\mathcal{L}}$ ∪ l; endif
□
```

a: sender's code

```
Initialization ⟶
    packets-set:=φ
    first_item:=true;
    packets-to-return := φ;
□
Receive$_{S\to R}$(l,i,packet) ⟶
    packets-set:=packets-set ∪ (l,i,packet)
    call check_and_output;
□
packets-to-return ≠ φ ⟶
    (l,i,packet):=extract(packets-to-return);
    Send$_{R\to S}$ (l,i,packet);
□
```

b: receiver's code

```
Procedure check_and_output
    if first_item=true and
        |packets-set|=Cap + 1 then
                                /* first data item */
        using the IDA calculate the data item from the
          Cap + 1 packets in packets-set;
        output(data-item);
        packets-to-return:=packets-to-return ∪ packets-set;
        packets-set:=φ;
        first_item:=false;
    else if first_item=false and
        |packets-set|=2·Cap + 1 then
                                /* all other data items */
          majority-label:=majority-of-labels(packets-set);
          using the IDA calculate the data item
                    from the packets in packets-set
                    having the label 'majority-label';
          output(data-item);
          packets-to-return:=packets-to-return ∪ packets-set;
          packets-set:=φ;
        endif
    endif
```

c: procedure check_and_output

Figure 3: The Data Dispersal Algorithm

this by requiring strong fairness in token distribution as outlined below.

Briefly stated, given the *slide* of Section 3, the generalized *slide* is implemented as follows: An additional pile, with a single slot at height n, is added to each *producer* node. This pile is managed in the same way as the special pile of the sender in the original *slide*. Whenever this pile is vacant, the producer can locally add a token to the *slide*. A *consumer* node is allowed to extract from its piles tokens for consumption; This is in addition to its ability to send tokens to adjacent nodes. Whenever it extracts a token from a pile it has to report this event to the other end of the link associated with this pile, as if the token was sent over a link. The strong fairness property is achieved as follows: If a node can forward tokens on a particular link infinitely often, then tokens will be forwarded on the link infinitely often. Furthermore, we assume that there is a conceptual link from a slot in a pile to the slot below it, which transfers tokens down if the slot is empty. This conceptual link is treated as a regular link as far as fairness goes. Finally, a token goes down only one slot when traversing any link; More than one token can temporarily reside at the same slot. However, since the rules for sending tokens over a physical link remain the same, the total number of tokens in a pile is still at most n. The mechanism described above is a version of *slide* that avoids starvation. We will use a generalized *slide* where all nodes are both consumers and producers. Note that the *slide* of Section 3 is a generalized *slide* with one producer and one consumer.

4.2 Combining the Algorithms

To build the combined algorithm we run two separate algorithms, a dynamic one and a static one, both receiving inputs and delivering outputs to a combination mechanism that outputs a single output for the problem being solved. We assume that this mechanism uses at most a constant number of output events of the dynamic and the static algorithms in order to produce one output event for the whole algorithm, and that it can continue to generate output events even if one of the two algorithms halts. There are three major steps in building the combined algorithm:

1. Whenever a node senses a topological change on one of its incident links it produces a *resource token*; A node running the dynamic algorithm has to consume such a token for each message it sends. Thus, each topological change creates at most 2 *resource tokens*, one per each node adjacent to the topological change. These two tokens can account for at most two messages sent in the dynamic algorithm. The generalized slide is employed to spread around the *resource tokens*. If the new *resource token* cannot be accepted by the *slide* because the node's pile is full, the *resource token* is discarded. This ties the dynamic algorithm to the topological changes.

2. A *reset* protocol, [AAG87], is used to tie the static algorithm to the topological changes. That is, the static algorithm should abort its execution if topological changes occur. Moreover, the *local* reset of [AAG87, AAM89] guarantees that topological changes in remote areas of the network do not unnecessarily affect the static algorithm.

3. We now have two independent algorithms running in the network. The question still stands how to distribute the inputs among them, and more importantly, how to merge their outputs into one correct output. The answer to this question is dependent on the particular problem and algorithms used; We discuss in the next section, as an example, the case of the end-to-end communication problem.

We first have to argue that the combined algorithm is correct, that is, eventually a new output event of either the static or the dynamic algorithm will occur: If the network topology stabilizes then the static algorithm never aborts and output events occur. If the network never stabilizes, then eventually nodes running the dynamic algorithm will have a token to consume and the algorithm will make progress.

For problems where one can build an algorithm that employs two totally independent algorithms, uses at most a constant number of outputs of each one to produce an output for the problem, and can operate even if one of the two halts, we have the following:

Theorem 1 *Given a dynamic algorithm $\mathcal{A}_d$ and a static algorithm $\mathcal{A}_s$ with communication complexity C_s per output event, there is an algorithm that has the following amortized communication complexity: $O(n+C_s+m)$ messages per topological change plus $O(C_s)$ per output event, where n and m are the number of nodes and links in the network, respectively.*

Proof Sketch. All messages sent by the combined algorithm are sent by one of the following four components: the dynamic algorithm $\mathcal{A}_d$, the static algorithm $\mathcal{A}_s$, the generalized *slide* and the *reset* mechanism of [AAG87].

The dynamic algorithm consumes a *resource token* to send each message, therefore sending at most $O(1)$ messages per topological change. Each token of the *generalized slide* makes at most n hops in the network, thus the *generalized slide* has $O(n)$ amortized message complexity per topological change. The amortized communication complexity of the reset mechanism [AAG87] is $O(m)$ per topological change. For each C_s messages of

the static algorithm either at least one output event of $\mathcal{A}_s$ occurs, or the output is purged because of a reset initiated by a topological change. Since after some constant number of output events of $\mathcal{A}_s$ there is an output event of the combined algorithm, any series of at most $O(C_s)$ messages of $\mathcal{A}_s$ can be charged to either an output event of the combined algorithm or to a topological change. □

4.3 Combined end-to-end

In this subsection, we outline the application of the combining methodology to create a dynamic end-to-end protocol that diverges into the static algorithm if the network is static for long stretches of time.

The problem of end-to-end communication from a sender to a receiver can be reduced to the problem of implementing two *probes* between these two processors, one in each direction (A probe implements the read of a remote variable); See [AG88]. Moreover, this paper shows how two independent probes in the *same* direction, can be merged into a single probe.

We use the combining methodology to solve the end-to-end problem as follows: We apply the methodology to static and dynamic probe algorithms, dispersing the inputs and merging the outputs as in [AG88], resulting in a combined probe (in one direction) from the sender to the receiver. Similarly, a corresponding combined probe is constructed in the opposite direction. The result is a pair of efficient probes, one in each direction, as is required (by the reduction of end-to-end to probe).

The static probe algorithm used is the obvious one–broadcast and echo along a fixed path. The dynamic algorithm is a flood forward and a flood back, in conjunction with the bootstrap mechanism of [AG91]. The end result is an end-to-end protocol whose amortized communication complexity is $O(n + m)$ messages per topological change plus $O(n)$ per data item. In particular, when the network topology stabilizes it achieves $O(n)$ message complexity.

Acknowledgments: We thank Baruch Awerbuch, Yishay Mansour, Michael Merritt, Michael Saks, Nir Shavit, and Gadi Taubenfeld for their cooperation and for helpful discussions.

References

[AAF+] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over unreliable channels.

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 358–370, October 1987.

[AAM89] Y. Afek, B. Awerbuch, and H. Moriel. A complexity preserving reset procedure. Technical Report MIT/LCS/TM-389, MIT, May 1989.

[AE86] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *NETWORKS*, 16(4):381–396, 1986. Previously titled "A Rigorous Treatment of a Communication Protocol: Broadcast as a Case Study.".

[AFWZ88] H. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. Reliable communication using unreliable channels. Manuscript, 1988.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliabel networks. In *Proc. of the Seventh ACM Symp. on Principles of Distributed Computing*, pages 131–148, August 1988.

[AG91] Y. Afek, , and E. Gafni. Bootstrap network resynchronization: An efficient technique for end-to-end communication. In *Proc. of the Tenth Annual ACM Symp. on Principles of Distributed Computing (PODC)*, August 1991.

[AGH90] B. Awerbuch, O. Goldreich, and A. Herzberg. A quantitative approach to dynamic networks. In *Proc. of the Ninth Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 189–203, August 1990.

[AM88] B. Awerbuch and Y. Mansour. An efficient topology update protocol for dynamic networks. Unpublished manuscript, January 1988.

[AMS89] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial end to end communication. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 358–363, October 1989.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 503–513, October 1990.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Verghese. Self-stabilization by local checking and correction. In *Proc. of the*

33rd IEEE Annual Symp. on Foundation of Computer Science, pages 268–277, October 1991.

[AS88] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proc. of the 29th IEEE Annual Symp. on Foundation of Computer Science*, pages 206–220, October 1988.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[BGP89] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 410–415, October 1989.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, January 1985.

[DF88] E. W. Dijkstra and W. H. J. Feijin. *A Method of Programming*. Addison-Wesley, 1988.

[DS80] W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11-1:1–4, August 1980.

[Fin79] S. G. Finn. Resynch procedures and fail-safe network protocol. *IEEE Trans. on Comm.*, COM-27:840–845, June 1979.

[MRR80] J. M. McQuillan, I. Richer, and E. C. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Communication*, COM-28(5), May 1980.

[MS80] P. M. Merlin and P. J. Schweitzer. Deadlock avoidance in store-and-forward networks 1: Store-and-forward deadlock. *IEEE Transaction on Communications*, 28(3):345–354, March 1980.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[Vis83] U. Vishkin. A distributed orientation algorithm. *IEEE Trans. Info. Theory*, June 1983.

[Wec80] S. Wecker. Dna: the digital network architecture. *IEEE Transactions on Communication*, COM-28:510–526, April 1980.

A The *slide* protocol

A.1 Correctness Proof

Throughout the proof we assume a global time, unknown to the nodes, and we denote the value of variables in a node at a given time by a subscript of the node and a superscript of the time (e.g. $\mathcal{X}_v^t$).

Definition 2 *$tokens\_intransit_{u\to v}^t$ is the number of tokens in transit from u to v at time t.*

Definition 3 *$signal\_intransit_{u\to v}^t$ is the number of SIGNAL messages in transit from u to v at time t.*

Lemma 4 *At any time t and for any e, $e=(u,v)$, $bound[e]_u^t - 1 = top[e]_v^t + tokens\_intransit_{u\to v}^t + signal\_intransit_{v\to u}^t$.*

Proof Sketch. By induction on the events that change any of the values participating in the equation.

Lemma 5 *A token received at node v from node u is stored at v in a slot at a lower level than the level it was stored in at u.*

Corollary 6 *Since new tokens enter the network into slot number n, every token can make at most n hops in the network.*

Lemma 7 *At any time t and for any e, $e=(u,v)$, $top[e]_v^t + tokens\_intransit_{u\to v}^t \le n$.*

Lemma 8 *At any time t there are no more than $2n$ messages in transit on any link.*

Note that the previous lemma shows that the protocol applies also in the model where links have constant (not just finite) capacity by having an $O(n)$ space buffer at the tail of each link and sending every message only after receiving an ACK for the previous one. As the space complexity of the protocol is anyhow $O(n)$, this change does not effect any of the complexities.

Lemma 9 *At any time t the number of tokens in the network is bounded by $O(nm)$. ($Cap = O(nm)$).*

Lemma 10 *In any time interval $[t_0,t_1]$, where* **new** *new tokens are added to the network, at most $O(n^2m+$**new**$\cdot n)$ token-passes can occur.*

Proof: By Lemma 9 the total number of tokens in the network at t_0 is bounded by $O(nm)$. By Lemma 6 each can make up to n hops in the network, thus contributing up to $O(n^2m)$ token passes. Any new token added in the time interval $[t_0,t_1]$ can also make up to n hops. □

Theorem 11 *If S and R are eventually connected, S will eventually input a new token.*

Proof: By way of contradiction assume that t is the last time the sender inputs a token.

As a result of Lemma 10 and as there is only one SIGNAL message per each token passing, there is a time $t' \geq t$ after which no token or SIGNAL message is sent. As S and R are eventually connected there is a path $R = v_0, v_1, \ldots, v_{k-1}, v_k = S$, $k < n$, such that for each $0 \leq i \leq k-1$, $e = (v_i, v_{i+1})$ is viable, hence there is a time $t'' \geq t'$ by which all messages between v_i and v_{i+1} have been delivered.

By induction on the length of the viable path from v_i to R, we can show that v_i cannot have a token in level $> i$ after time t'' .

The receiver, v_0, has no tokens stored at all. Denote by e the link between v_i and v_{i-1} ($i \geq 1$), and assume the induction assumption that v_{i-1} has no token stored at level $> i-1$. Since at t'' all messages between v_{i-1} and v_i have arrived, by Lemma 4 $bound[e]^{t''}_{v_i} = top[e]^{t''}_{v_{i-1}} + 1$; by the induction assumption $top[e]^{t''}_{v_{i-1}} \leq i-1$, thus $bound[e]^{t''}_{v_i} \leq i$. As $t'' \geq t'$, no token is sent after t'', but according to the code this can happen only if v_i does not have tokens of level $i+1$ or more, proving the induction step.

Since $k < n$, S does not have a token of level n at t'', and by the code will enable input of a new token, contradicting the assumption that t is the last time S inputs a token. □

A.2 Complexity

Lemma 12 *The number of messages sent by* slide *in any time interval where* **new** *new tokens are input by the sender is bounded by* $O(n^2m + \mathbf{new} \cdot n)$.

Corollary 13 *The number of bits sent in the* slide *protocol in any time interval where* **new** *new tokens are input by the sender is bounded by* $O(n^2m + \mathbf{new} \cdot n)L)$, *where L is the maximal number of bits in a token.*

Lemma 14 *The space needed at each node is* $O(nL)$ *per incident link, where L is the maximal number of bits in a token.*

B The Majority Algorithm

B.1 Correctness Proof

Definition 15 $in^{[t,t']}$ *is the number of tokens deposited by the sender into the* slide *in the time interval* $[t, t']$.

Definition 16 $out^{[t,t']}$ *is the number of tokens received by the receiver from the* slide *in the time interval* $[t, t']$.

Definition 17 $delay^t$ *is the number of tokens delayed by the* slide *at time t.*

Theorem 18 (Safety) *At any time the output of the receiver is a prefix of the input of the sender.*

Proof: We denote by $I = \{I_1, I_2, \ldots\}$ and by $O = \{O_1, O_2, \ldots\}$ the input to the sender and the output of the receiver, respectively. Denote by t_0 some time before the beginning of the execution of the algorithm, and by $t_i, i > 0$ the time at which O_i is output.

To prove the theorem, we claim that the majority of the tokens received by the receiver in the interval of time $[t_{i-1}, t_i]$ carry data item I_i. First we show that no token carrying I_k, $k > i$ could have been received before t_i. By the code, the total number of tokens that have been received by the receiver until time t_i is:

$$out^{[t_0,t_i]} = Cap + 1 + (i-1)(2 \cdot Cap + 1).$$

Since the network capacity is Cap, the total number of tokens sent by the sender at any time t is at most Cap more than the total received by the receiver at the same time, t. Thus,

$$in^{[t_0,t_i]} \leq i(2 \cdot Cap + 1) \qquad (1)$$

Therefore, no token carrying I_k, $k > i$ can be sent by the sender before t_i. Hence, no such token can be received by the receiver at t, $t < t_i$.

We claim that no more than Cap tokens containing data item I_k, $k < i$ may be received in the interval of time $[t_{i-1}, t_i]$. This completes the proof of the safety property because together with the above it implies that from the $2 \cdot Cap + 1$ tokens received in $[t_{i-1}, t_i]$ at least $Cap + 1$ of them carry data item I_i.

To prove the claim we distinguish between two sets of tokens, those that carry data items I_k, $k < i$, which we call *old*, and the all other tokens. We have already proved that all the tokens received until t_{i-1} are *old* and that the total number of such tokens received by the receiver until t_{i-1} is $(2 \cdot Cap + 1)(i-1) - Cap$. Since the total number of *old* tokens ever sent by the sender is $(2 \cdot Cap + 1)(i-1)$, at most Cap may be received by the receiver in the interval of time $[t_{i-1}, t_i]$. □

Theorem 19 (Liveness) *If the sender and the receiver are eventually connected, then the receiver will eventually output any data item input by the sender.*

B.2 Complexity

Lemma 20 *The message complexity of the majority algorithm is* $O(n^2m)$.

Proof: Let T_i be the interval of time from the time O_{i-1} is output to the time O_i is output. Clearly in T_i the receiver receives $2 \cdot Cap + 1$ tokens. Since the *slide* can hold at most Cap tokens, at most $3 \cdot Cap + 1$ tokens are sent by the sender in T_i, and the lemma follows. □

Corollary 21 *The bit communication complexity of the majority algorithm is $O(n^2mD)$, where D is the size in bits of a data item.*

Lemma 22 *The space complexity of the majority algorithm is $O(nD)$, where D is the size in bits of a data-item.*

Proof: Each token sent in the majority algorithm consists of $O(D)$ bits. Applying this to Lemma 14, we get the space complexity of $O(nD)$ bits. □

C The Data Dispersal Algorithm

C.1 Correctness Proof

Lemma 23 *Whenever the sender tries to extract a label from $\overline{\mathcal{L}}$, $\overline{\mathcal{L}}$ is not empty.*

Theorem 24 (Safety) *At any time the output of the receiver is a prefix of the input of the sender.*

Proof: We denote by $I = \{I_1, I_2, \ldots\}$ and by $O = \{O_1, O_2, \ldots\}$ the input to the sender and the output of the receiver, respectively. Let t_i be the time when the receiver outputs the i'th data item, and denote by l_i the label added to the $2 \cdot Cap + 1$ packets calculated from I_i at the sender. By the code, the tokens used at t_i to calculate the i'th data item at the receiver are the $2 \cdot Cap + 1$ tokens received by it in the time interval $[t_{i-1}, t_i]$. By the same arguments used for the safety proof of the majority algorithm (Theorem 18), at least $Cap + 1$ of these tokens contain the label l_i; thus the majority of labels will be l_i, and the receiver will calculate the i'th data item from the tokens containing l_i. Since at the time the sender extracts l_i from $\overline{\mathcal{L}}$ there is no token containing it in the network, the IDA at the receiver at t_i will use only packets calculated from I_i at the sender. As noted before the receiver has at least $Cap + 1$ such packets at t_i, and the IDA will correctly calculate I_i at t_i. Thus $O_i = I_i$ for any i. □

Lemma 25 *For any time t missing $\leq 4 \cdot Cap + 1$.*

Proof: The variable *missing* is incremented when a token is extracted from *send_buffer*. The only event where tokens are added to *send_buffer* is when $2Cap+1$ tokens are added to it when it is empty and *missing* $\leq 2 \cdot Cap$. Thus at any time *missing* $\leq 4 \cdot Cap + 1$. □

Note that this also implies that the receiver never stores more than $4 \cdot Cap + 1$ tokens in all its buffers.

Lemma 26 *If the sender and the receiver are eventually connected, there is no dead-lock at the sender (eventually missing $\leq 2 \cdot Cap$).*

Theorem 27 (Liveness) *If the sender and the receiver are eventually connected, then the receiver will eventually output any data item input by the sender.*

C.2 Complexity

Lemma 28 *The message complexity of the data dispersal algorithm is $O(n^2m)$ messages.*

Proof: Denote by t_i the time the i'th data item is output at the receiver. We use for the *slide* from the sender to the receiver the same notation as in Section B.1. By the code $out^{[t_i,t_{i+1}]} = 2 \cdot Cap + 1$, and $0 \leq delay^i \leq Cap$, thus $Cap + 1 \leq in^{[t_i,t_{i+1}]} \leq 3 \cdot Cap + 1$. By Lemma 9 $Cap = O(nm)$, and applying this to Lemma 12 yields a message complexity of $O(n^2m)$ for the *slide* from the sender to the receiver.

The tokens that are sent through the *slide* from the receiver to the sender in the time interval $[t_i, t_{i+1}]$ must be in *tokens_to_return* just after t_i, since new tokens are added to this set only at output events at the receiver. By Lemma 25 the receiver stores at any time at most $4 \cdot Cap + 1$ tokens. In the worst case all of them are in *tokens_to_return* at t_i. Thus at most $4Cap+1$ tokens are input to this *slide* in the time interval $[t_i, t_{i+1}]$. Applying this to the results of Lemmas 9 and 12 we obtain a message complexity of $O(n^2m)$ for this *slide*.

Combing the two *slide* protocols we obtain a message complexity of $O(n^2m)$ for the data dispersal algorithm. □

Corollary 29 *The bit communication complexity of the data dispersal algorithm is $O(nD)$, where D is the size in bits of a data item.*

Proof: Each token sent in the data dispersal algorithm consists of a packet of size $O(\frac{D}{Cap+1})$, a label of size $O(\log n)$, and a serial number of size $O(\log n)$. The message complexity of the algorithm is $O(n^2m)$, and half of the messages are of size $O(\frac{D}{Cap+1} + \log n)$, while the other half have constant size. The total number of bits sent between any two consecutive output events at the receiver is, therefore, $O((n^2m)(\frac{D}{Cap+1} + \log n))$. Since $Cap = O(nm)$ this is $O(nD + n^2m \log n)$. For the data dispersal algorithm the data item is of size $\Omega(nm \log n)$, therefore the bit complexity is $O(nD)$. □

Lemma 30 *The space complexity of the data dispersal algorithm is $O(\frac{D}{m} + n \log n)$, where D is the size in bits of a data item.*

Proof: Each token sent in the algorithm is of size $O(\frac{D}{Cap+1} + \log n)$. By applying Lemma 14 we get the space complexity of $O(n(\frac{D}{Cap+1} + \log n))$. Since $Cap = O(nm)$ the space complexity is $O(\frac{D}{m} + n \log n)$. □

Computing with Faulty Shared Memory

(Extended Abstract)

Yehuda Afek* David S. Greenberg† Michael Merritt‡ Gadi Taubenfeld‡

Abstract. This paper addresses problems which arise in the synchronization and coordination of distributed systems which employ unreliable shared memory. We present algorithms which solve the consensus problem, and which simulate reliable shared-memory objects, despite the fact that the available memory objects (e.g. read/write registers, test-and-set registers, read-modify-write registers) may be faulty.

1 Introduction

Research on fault-tolerant, shared-memory systems typically studies processor failures and assumes that the shared memory is reliable [Lam86, Her91, BP87, VA86, Blo87, SAG87, CIL87, Abr88, Plo88, Rab82, AAD+90]. However, memories do fail. For example, a shared register might, after failing, return some arbitrary value to subsequent read operations. This paper investigates the effects of shared memory failures in distributed systems.

One common technique for dealing with faulty memory is to keep many copies of each datum or, in general, to use some type of redundant coding which allows errors to be detected and/or corrected. But if the redundancy is included in the value of a single register, then an arbitrary spontaneous change of the entire contents of the register will not be detected. Additionally, the increased size of the register makes it more difficult to build – and perhaps more prone to failure.

In a shared memory environment these techniques are less useful. A shared memory for a distributed system is much more complex to implement than the memory of a uni-processor. Common primitives such as read-modify-write or even simple reads and writes require a memory which is much more than a passive repository of data (see, e.g. [Smi82]). If redundancy is spread across registers then the encoding is subject to the timing inconsistencies which are the bane of all distributed algorithms. Strategies for rectifying shared memory faults will have to incorporate distributed coordination techniques. Hence, the need arises to understand the power of memories, some of whose cells may be faulty. Algorithms resistent to memory faults can be used directly to create reliable applications. Alternatively, a reliable shared memory can be implemented from unreliable memories, thereby providing reliable primitives to applications which cannot tolerate faulty memory.

We show that both these approaches are viable. After a discussion of memory failures and specification of faulty memory primitives (in Section 2), the body of this paper presents a sequence of algorithms that use faulty shared-memory. Section 3 begins with algorithms for implementing reliable read/write memories from faulty memories. Following Lamport [Lam86], we present simple constructions of safe and regular registers and conclude with a construction of a reliable atomic register from faulty atomic registers.

We turn next to studying the consensus problem in shared memory models, using unreliable versions of the more powerful primitives *test-and-set* (Section 4) and *read-modify-write* (Section 5). The algorithms we present demonstrate that faults do not qualitatively decrease the power of these primitives, in that they retain their positions in the memory hierarchy of Herlihy [Her91]. Moreover, in combination with the earlier register results, our consensus algorithms can be used to implement the universal construction of Herlihy [Her91]. Hence, for example, faulty read-modify-write primitives can be used to implement any shared object. The paper closes with a discussion of related work and open problems.

2 The model

Perhaps the most general memory fault is a spontaneous change in the value of a register. Certainly if every register can spontaneously change to a new value at any time then the situation is hopeless. Our goal is to explore how much of the memory can be faulty while still allowing a specific problem to be solved, or a fault-free memory to be simulated. We will therefore restrict the number

*Computer Science Department, Tel-Aviv University, Israel 69978, and AT&T Bell Laboratories.

†Sandia National Laboratories, Mail Stop 1423, P.O. Box 5800 Albuquerque, NM 87185-5800. Supported in part by the U.S. Department of Energy under contract DE-AC04-76DP00789.

‡AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

PoDC '92-8/92/B.C.

© 1992 ACM 0-89791-496-1/92/0008/0047...$1.50

of memory objects which can change and/or the number of times they can change. Other less general types of faulty behaviors might include registers which never change their value (after becoming faulty all reads return the same value regardless of any write command), registers which occasionally miss a write (the written value is never available to a read), and registers which occasionally return the wrong value (some reads return arbitrary values or return values which are not consistent with any ordering of the writes). Alternatively, the timing of the faults might be restricted; for example, all memory faults might occur before any processor takes a step.

2.1 Specifying faulty memory

We consider a collection of asynchronous processors which communicate via a shared memory. The shared-memory may consist of a variety of shared data objects. This shared memory is subject to *memory failures*, each modeled as a write to a shared data object which is atomic with respect to the processors' operations on that object. (The local, non-shared memory for each processor is assumed to be reliable. Error-correcting codes and similar redundant techniques may be employed in local memories, where issues of communication, synchronization and processor failures are less critical.)

Faulty variants of atomic data objects are specified as follows. Each *reliable* object X has a type, which defines a set of possible *values*; a set of primitive operations, which provide the only means to manipulate that object; and a set of runs, which constitute the *sequential specification* of X and define how the object behaves when its operations are invoked one at a time [HW90]. A *faulty object*, X, extends the set of operations with a set of write(v) operations (for any v which is ever a legal value of X) invocations of which constitute *failures* of X. The sequential specification of the faulty object now includes failures of X. Provided serial operations on a reliable X are total, the sequential specification of a faulty object is a function of that of the reliable object, in the obvious way. (If serial operations on a reliable X are not defined on some values, a definition of the faulty serial specification, specific to X, may be required.) Given the sequential specification, a complete concurrent specification may be derived using any of a number of techniques or styles [Lam86, HW90, VA86, AAD+90, BP87, Blo87, Her91, LT87].

Some widely-studied objects are not atomic—the regular and safe registers defined by Lamport are the best-known examples [Lam86]. In these particular instances, write operations are already defined on the objects, and it is straightforward to extend their specification to include faulty write operations in addition to those invoked by processors.

As observed above, some constraints must be imposed on the occurance of memory faults:

- We use m to denote the total number of memory failures (faulty write operations) in a run (or collection of runs) of a system.
- We use f to denote the total number of data objects in a system that may be affected by memory failures in a run (or collection of runs) of a system.
- A data object is *k-faulty* if it suffers at most k failures, i.e., at most k faulty write operations to it are invoked during each run. A data object is *∞-faulty* if there is no finite bound on the number of failures it suffers.

In algorithms working with faulty shared memory, we generally require the algorithms to be *strongly-wait-free*; any operation by a processor must terminate its execution, regardless of the number of shared memory faults and independent of the steps taken by other processors. Thus, a strongly-wait-free algorithm may correctly implement a shared object for only a bounded number of memory faults, but each high-level operation by a non-faulty processor must still terminate, even if the bound on the number of memory faults is exceeded in a given run.

In order to quantify the cost of having a certain number and type of memory faults, we define a function, CONS, which represents the number of copies of one object, some of which are faulty, that are necessary to construct another object:

Definition

$\mathbf{CONS}(X, m, Y) \stackrel{\text{def}}{=}$ the number of objects of type X required to construct one non-faulty object of type Y, assuming there may be at most m memory faults among the type X objects.

$\mathbf{CONS}_k(X, f, Y) \stackrel{\text{def}}{=}$ the number of objects of type X required to construct one non-faulty object of type Y, assuming at most f of the type X objects may be k-faulty (k can be ∞).

2.2 A general construction

We start with a simple theorem in which we show how a solution that tolerates one faulty register can be improved to tolerate f faulty registers.[1]

Theorem 1 *For any* $f > 1$, $\mathbf{CONS}_\infty(X, f, X) \leq (2f)^{\log \mathbf{CONS}_\infty(X,1,X)} = (\mathbf{CONS}_\infty(X, 1, X))^{1+\log f}$.

[1] Throughout, logarithms are base 2 unless otherwise noted.

Proof: Assume there is a construction of a reliable object of type X, using $C = \mathbf{CONS}_\infty(X, 1, X)$ strongly-wait-free objects of type X, one of which may be ∞-faulty.

The goal is to construct a strongly-wait-free object X that is reliable, even if f of the component registers are ∞-faulty. That is, to construct a reliable object X from a set of objects X, f of which might be faulty. Consider the following recursive construction: we first construct C strongly-wait-free objects of type X, each of which is resilient to $\lfloor f/2 \rfloor$ faulty registers, then we use these C objects to construct a single object X, using the single fault construction. The result is an object that can tolerate the failure of one of the embedded $\lfloor f/2 \rfloor$-memory resilient objects. For one of the embedded objects to fail, there must be at least $\lceil f/2 \rceil$ memory faults in it. Since the total number of faults is $f = \lceil f/2 \rceil + \lfloor f/2 \rfloor$, none of the other embedded objects is faulty. Hence, the final construction is tolerant of at least f memory faults. The total number of X objects used in this construction is: $\mathbf{CONS}_\infty(X, f, X)$

$$\begin{aligned} &= \mathbf{CONS}_\infty(X,1,X) \cdot \mathbf{CONS}_\infty(X, \lfloor f/2 \rfloor, X) \\ &\leq \mathbf{CONS}_\infty(X,1,X)^{\lfloor \log f \rfloor + 1} \\ &\leq \mathbf{CONS}_\infty(X,1,X)^{\log f + 1} \\ &= (2f)^{\log \mathbf{CONS}_\infty(X,1,X)}. \end{aligned}$$

∎

A similar recursive construction can be based on any self-construction resilient to $c \geq 1$ register faults. Moreover, the same construction works for weaker types of failure:

Theorem 2
For any f and c, $f > c > 0$, and for all $k \in \{1, ...\} \cup \{\infty\}$,

$$\begin{aligned} \mathbf{CONS}_k(X,f,X) &\leq ((c+1)f)^{\log_{c+1} \mathbf{CONS}_k(X,c,X)}; \\ \mathbf{CONS}(X,m,X) &\leq ((c+1)m)^{\log_{c+1} \mathbf{CONS}(X,c,X)}. \end{aligned}$$

Fault-tolerant constructions can be composed with fault-intolerant constructions:

Theorem 3

$$\begin{aligned} \mathbf{CONS}_k(X,f,Z) &\leq \mathbf{CONS}_k(X,f,Y) \cdot \mathbf{CONS}_k(Y,0,Z); \\ \mathbf{CONS}(X,m,Z) &\leq \mathbf{CONS}(X,m,Y) \cdot \mathbf{CONS}(Y,0,Z). \end{aligned}$$

3 Read/Write registers

One approach to tolerating faulty registers is to add a software layer between the faulty hardware and the user which looks to the user like fault-free hardware. In this section we show that this is possible for safe, regular, and atomic registers. That is, we present constructions of safe, regular, and atomic registers from a collection of the corresponding primitives, f of which may be ∞-faulty.

Theorem 4 *One reliable, strongly-wait-free, safe register can be constructed from $2f+1$ similar registers, f of which may be ∞-faulty:* $\mathbf{CONS}_\infty(safe, f, safe) = 2f+1$.

Proof: For the upper bound, the obvious construction works: the writer writes the $2f+1$ registers, and the reader reads them. If the reader sees a majority value ($f+1$ with the same value), it returns that value, otherwise it returns any value. The lower bound is similar to the proof of Theorem 11. (Note that this result holds for multi-reader/multi-writer safe registers. In what follows, registers are assumed to be single-reader/single-writer.) ∎

Since a single (reliable) safe bit is sufficient to implement a regular bit [Lam86], it follows from Theorem 4 that $\mathbf{CONS}_\infty(binary\_safe, f, binary\_regular) = 2f + 1$. Moveover, given Theorem 4, one can construct any (multi-reader/multi-writer, arbitrary value) atomic register using constructions from safe bits [Pet83, Lam86, BP87, PB87, Blo87, SAG87, LTV89, Tro89]. For example, a construction due to Tromp [Tro89] produces a binary atomic register from 3 safe bits (3 are necessary [Lam86]),

$$\begin{aligned} &\mathbf{CONS}_\infty(binary\_safe, f, binary\_atomic) \\ &\quad \leq \mathbf{CONS}_\infty(binary\_safe, f, binary\_safe) \\ &\qquad \cdot \mathbf{CONS}_\infty(binary\_safe, 0, binary\_atomic) \\ &\quad = 6f + 3. \end{aligned}$$

Define a V-register to be a read/write register on (arbitrary) value domain V. A (fault-intolerant) construction by Peterson [Pet83] produces an atomic V-register from 3 safe V-registers and 4 atomic binary registers. Another construction due to Tromp [Tro89] produces an atomic V-register from 4 safe V-registers and 8 safe binary registers. Both these constructions can be composed with the construction in Theorem 4, to construct reliable atomic V-registers from unreliable safe V-registers and unreliable safe bits. The construction in Figure 1 constructs a reliable atomic V-register directly from unreliable components, specifically, $8f + 2$ atomic V-registers and 2 reliable atomic bits. (The proof of this construction is omitted from the extended abstract.) Using Theorem 4 and the constructions just discussed of atomic bits from safe bits, we have:

Theorem 5 *One reliable, strongly-wait-free, atomic V-register can be constructed from the combinations of components below, where in each case, up to f of the combined components may be ∞-faulty:*

- $6f + 3$ *safe V-registers and* $24f + 12$ *safe binary registers ([Pet83] and Theorem 4).*
- $8f + 4$ *safe V-registers and* $16f + 8$ *safe binary registers ([Tro89] and Theorem 4).*
- $8f + 2$ *atomic V-registers and* $12f + 6$ *safe binary registers (Figure 1).*

```
R, W: reliable atomic on {0,1}
val[0..1, 1..4f+1]: 2-D array, atomic in Value, unreliable

Writer's protocol:
function write(v)                          % v is a value
ptr := ¬R              % copy not being used by reader
for i = 1 to 4f+1 do val[ptr, i] := v od
W := ptr               % tell future readers where to look
end_function

Reader's protocol
function read: value
sawW, prev: persistent across invocations
                          % initially false and 0, resp.
newW := W                  % where writer last finished
if R = newW and sawW       % no evidence of a write
  then return(prev)        % return last value read
else R := newW             % read copy last writen
  sawW := false                        % remember state
  for i = 4f+1 down to 1 do
    tmp[i] := val[R, i]
  if a^{f+1} b^{f+1}, a ≠ b, is a subseq. of tmp[1..4f+1]
  then              % must have overlapped a write(a)
    sawW := true                       % remember state
    prev := a              % return a until state changes
    return(a)
  else       % don't know value of a concurrent write
    return(the majority value in tmp[1..4f+1])
end_function
```

Figure 1: A reliable atomic register

If we consider the construction of atomic registers *only* from faulty atomic registers of the same type, the construction from Figure 1 dominates (using V-registers to implement safe bits):

Corollary 6 $\mathbf{CONS}_\infty(atomic, f, atomic) \leq 20f + 8$.

4 Test-and-set registers

Unfortunately, atomic registers do not provide a very strong memory primitive. Even the simple task of two-processor consensus is impossible with just atomic registers. Such tasks require a stronger primitive such as test-and-set.

A *two-processor, binary, test-and-set register* is a concurrent object accessible by two processors through the operations test&set and reset. The sequential specification of the object is most simply understood as operations on a binary register, initialized to 0. The test&set operation atomically reads the register, writes 1 into it, and returns the value read. The reset operation writes 0. If the object is faulty, the failure operations write(0) and write(1) have the obvious effect.

The processors are constrained in their use of the reset operation – a processor should only invoke the reset operation if its previous operation on the object was a test&set that returned 0. If the processors violate this well-formedness condition then the object may exhibit arbitrary behavior.

A *single-use* test-and-set has no reset operation. (Except in the statements of the theorems, we will say "test-and-set" instead of "two-processor, binary test-and-set".)

In this section we show that reliable test-and-set registers can be constructed from unreliable test-and-set registers. We will allow the test-and-set constructions to utilize reliable atomic read/write registers. (These weaker but reliable registers can, of course, be constructed from a set containing unreliable components using the techniques of the last section.) We first show the constructions for single-use test-and-set and then extend to multi-use.

Theorem 7

- *There is a strongly-wait-free, two-processor, single-use, binary, test-and-set algorithm using 7 binary test-and-set registers, 1 of which may be ∞-faulty (Figures 2 and 3).*
- *There is a strongly-wait-free, two-processor, binary, test-and-set algorithm using 14 binary test-and-set registers, 1 of which may be ∞-faulty, and 4 reliable, binary, atomic read/write registers. (Figure 4).*

Corollary 8 *There is a strongly-wait-free, two-processor, binary, test-and-set algorithm using $14f^{\log 14}$ test-and-sets, f of which may be ∞-faulty, and $4(2f)^{\log 14} = 56f^{\log 14}$ reliable, binary, atomic registers.*

```
Protocol for processor p:              % Other proc. is q.
shared A[1..3], B, C[1..3]: T&S objects, initially 0

function s-test&set        % return win (0) or lose (1)
sum := 0
for i = 1 to 3 do
    sum := sum + test&set(A[i])
if sum < 2 then goto C                          % p won A
else if test&set(B)= 0 then return(1)
    else
C:      sum := 0
        for i = 1 to 3
            do sum := sum + test&set(C[i])
        if sum < 2 then return(0)               % p won C
        else return(1)                          % p lost C
end_function
```

Figure 2: Single-use test-and-set.
Uses 7 test-and-set's, 1 of which may be faulty.

Since two-processor consensus can be implemented with three single-use test-and-set registers [LA87], we have:

$\mathbf{CONS}_\infty(single\_use\_T\&S, 1, two\_consensus)) \leq 21$.

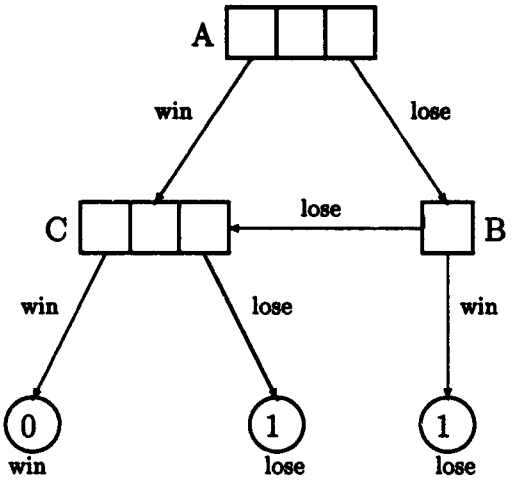


Figure 3: Schematic of single-use test-and-set (from Figure 2).

```
Protocol for processor p:                    % Other proc. is q.
shared TS[0..1]: single-use t&s objects
                 with s-test&set operation
    % written only by winner, initially 0
    lose, current: reliable atomic on {0,1}
    % written only by owner, initially 0
    my_current[p..q]: reliable atomic on {0,1}

function test&set
t := current
my_current[p] := t
if lose then return(1)
else return(s-test&set_t)
end_function

function reset
lose := 1                          % Prevent q from winning
other := my_current[q]                    % where is q?
reset all of TS[¬other]                % clean other copy
current := ¬other                % Use clean copy in future
lose := 0                      % Done reset, allow q freedom
end_function
```

Figure 4: Multi-use binary test-and-set.

Moreover, mutual exclusion is possible by using multi-use test-and-sets as critical section locks. However, binary test-and-set is too weak to implement consensus for more than 2 processors [LA87].

Theorem 7 follows from the constructions in Figures 2, 3 and 4. (Details are omitted from this extended abstract.)

5 RMW registers and consensus

To complete our investigation of memory fault-tolerance we turn to an object from the top of Herlihy's hierarchy of shared memory objects, the read-modify-write register. Herlihy showed that *reliable* read-modify-write (RMW) is a universal shared-memory primitive [Her91]. Any other shared object can be simulated using RMW. This follows because RMW registers can be used to solve n-processor consensus, which is itself universal.

Briefly, RMW registers enable a processor to atomically read a register and, *based on the value read,* to write a new value. In the consensus problem there are n processors, each with an input value, $input_p \in \{-1, +1\}$. A processor *decides* on a value *output* if it writes *output* to its write-once output register. The requirements of the consensus problem are that there exist a *decision value* v such that each non-faulty processor eventually decides on v and that v is the input value of at least one processor.

5.1 Bounded failures per register

For a bounded number of faults per register we are able to fully characterize the number of RMW registers required for consensus:

Theorem 9 *For any $m \geq 0$, there is a strongly-wait-free consensus algorithm using $2m+1$ faulty read-modify-write registers, provided the total number of memory failures is at most m:*

$$\text{CONS}(RMW, m, consensus) \leq 2m+1.$$

Recall that a register is k-faulty if it can change its value spontaneously, without any processor writing into it, at most k times.

Corollary 10 *For any $1 \leq k \leq m$, there is a strongly-wait-free consensus algorithm using $2m+1$ RMW registers where at most $\lfloor \frac{m}{k} \rfloor$ registers are k-faulty:*

$$\text{CONS}_k(RMW, \left\lfloor \frac{m}{k} \right\rfloor, consensus) \leq 2m+1.$$

For $k = 1$, the above bound is tight:

Theorem 11 *There is no n-processor consensus algorithm using fewer than $2f+1$ RMW registers, at most f of which may be 1-faulty, and which survives $\lceil n/2 \rceil$ processor failures.*

Proof: Assume to the contrary that there is a solution using $2f$ registers. Let the initial values of the registers $r_1, ..., r_{2f}$, be $u_1, ..., u_{2f}$, respectively.

The processor-failure assumption requires that in any run in which only half the processors, $p_1, ..., p_{\lceil n/2 \rceil}$ take steps, they must eventually decide. Also, if their inputs are identical, the validity condition requires that they must decide on that value, as they don't know whether the other decision value is an input of some processor.

Thus, there is a failure-free, finite run x where half the processors run alone with input $+1$ and decide on $+1$. Let the final values of the registers $r_1, ..., r_f$, in x be $v_1, ..., v_{2f}$, respectively.

Now consider a run in which the other processors, $p_{\lceil n/2 \rceil+1}, ..., p_n$ take steps, run alone with input -1, but find the registers hold values $u_1, ..., u_f, v_{f+1}, ..., v_{2f}$. This is consistent with a run in which $p_1, ..., p_{\lceil n/2 \rceil}$ have taken no steps and f faults have changed the values of $r_{f+1}, ..., r_{2f}$ to $v_{f+1}, ..., v_{2f}$, respectively. Hence, $p_{\lceil n/2 \rceil+1}, ..., p_n$ must all decide -1. But this run is also consistent with a run in which $p_1, ..., p_{\lceil n/2 \rceil}$ have already decided $+1$, and f faults have changed the values of $r_1, ..., r_f$ back to $u_1, ..., u_f$, respectively. Hence, $p_{\lceil n/2 \rceil+1}, ..., p_n$ must all decide $+1$, a contradiction. ∎

Corollary 12

- $\mathbf{CONS}_1(RMW, f, consensus) = 2f+1$, *and*
- $\mathbf{CONS}(RMW, m, consensus) = 2m+1$.

The proof of Theorem 9 is based on the algorithm in Figure 5, which solves strongly-wait-free consensus using $2m+1$ faulty read-modify-write registers, provided the total number of memory failures is at most m. The input to processor p is initially assigned to register $input_p$, local to p, and an appropriate output values must be assigned to register $decide_p$. Figure 5 uses the notation **lock**(r) and **unlock** to mark the beginning and end of atomic, exclusive access to shared read-modify-write register r. That is, it is assumed that a processor can lock only one register at a time, that a processor does not fail between pairs of **lock**(r) and **unlock** statements, and that any non-faulty processor that reaches a **lock** instruction eventually executes it.

The proof of correctness proceeds as follows. Note first (from lines 5–10 in the protocol body) that each processor's protocol performs $O(m^2)$ wait-free operations before deciding and terminating. Hence, the algorithm is strongly-wait-free, and it suffices to consider only *complete runs*, those in which every processor has terminated. Thus, the algorithm is correct if its complete runs satisfy the validity and agreement conditions. The algorithm is analyzed under a stronger fault model which allows m independent faults to occur to each of the *vote*, *plus* and *minus* fields of the shared registers, up to $3m$ faults in all. These faults are modeled as assignments to the appropriate register fields. The validity condition is proven in a straightforward manner (Lemma 5.1). Next, we argue that the algorithm is correct if and only if each of a constrained set of executions is correct (Lemmas 5.2-5.4). These executions are shown to satisfy an invariant that implies the agreement condition (Lemmas 5.5-5.7).

Note that nowhere in any processor's code is a shared register field ever set to 0.

Protocol for processor p, $input_p \in \{-1, +1\}$:

```
type  reg = record minus, plus: in {0,1}
                   vote: in {-1,0,+1}
shared r: array[1..(f+1)^2] of reg, initially (0,0,0)
local decide_p: in {-1,0,+1}, initially 0

1:  for i = 1 to 2m+1 do
                % indicate that input_p is a valid input
2:      if input_p = +1 then RMW(r[i], plus, 1)
3:      else RMW(r[i], minus, 1) od
4:  d := input_p ;
5:  for i = 1 to 2m+1 do
                % push sum away from 0
6:      RMW(r[i], vote, d)
7:      sum := 0
8:      for j = 1 to i do
9:          sum := sum + RMW(r[j], vote, d) od
10:     d := valid(sum) od
11: decide_p := d ;              % make final decision

1:  function RMW(reg, field, t): integer
            % atomically set reg.field from 0 to t.
2:  lock(reg)
3:      if reg.field = 0 then reg.field := t
4:      tmp := reg.field
5:  unlock
6:  return(tmp)
7:  end_function

1:  function valid(v): integer
          % return sign(v) if valid, input_p otherwise
2:  if v = 0 or sign(v) = sign(input_p)
3:  then return(input_p)
4:  else sm := 0
5:      for j = 1 to 2m+1 do
6:          if sign(v) = +1 then sm := sm + r[j].plus
7:          else sm := sm + r[j].minus
8:      od
9:      if sm ≤ m
10:     then return(input_p)          % v is not valid
11:     else
12:         for j = 1 to 2m+1 do
13:             if sign(v) = +1
14:             then RMW(r[j], plus, 1)
15:             else RMW(r[j], minus, 1)
16:         od
17:         return(sign(v))
18: end_function

1:  function sign(x): integer
2:  if x > 0 then return(+1)
    else if x = 0 then return(0)
    else if x < 0 then return(-1)
3:  end_function
```

Figure 5: Consensus in the presence of m memory failures.

Lemma 5.1 (Validity)

1. *No processor p writes to a plus (respectively, minus) field unless either $input_p = +1$ (respectively, $input_p = -1$), or the processor has previously observed 1 as the value in a $m+1$ of the plus (respectively, minus) fields.*

2. *No processor writes to a plus (respectively, minus) field unless $+1$ (respectively, -1) is the input of some processor.*

3. *No processor decides $+1$ (respectively, -1) unless the processor has previously observed 1 as the value in $m+1$ of the plus (respectively, minus) fields.*

4. *No processor writes $+1$ (respectively, -1) to a vote field without first assigning 1 to the plus (respectively, minus) fields of all $2m+1$ registers.*

We call two complete runs x and y *similar* if each processor has the same input value in x as in y, and decides on the same value in x as in y.

Next, note that the read-modify-write in line 9 modifies $r[j].vote$ only if $r[j].vote$ is first observed to be 0. Since the same processor will have either set or observed $r[j].vote \neq 0$ in an earlier scan, this observation of 0 and resulting modification is due to a memory fault on $r[j].vote$.

Lemma 5.2 *For any complete run y there is a similar run x, such that x contains no more memory faults than y, and such that in x no memory fault assigns 0 to any vote field.*

Proof: Let y be a complete run of the form $y_1; r[j].vote := 0; y_2$, where $r[j].vote := 0$ is a memory fault. There are several cases:

- No operation in y_2 references $r[j].vote$.
  Then $y_1; y_2$ is a complete run that is similar to y, has no more memory faults than y, and has one fewer (faulty) assignments of 0 to a *vote* field.

- The first reference to $r[j].vote$ in y_2 is a memory fault. Then y_2 can be written as $y_3; r[j].vote = v; y_4$, where y_3 contains no reference to $r[j].vote$, $r[j].vote = v$ is a memory fault and $v \in \{-1, 0, 1\}$. Then $y_1; y_3; r[j].vote := v; y_4$ is a run of the algorithm that is similar to y, contains no more memory faults than y, and has one fewer (faulty) assignments of 0 to a *vote* field.

- The first reference to $r[j].vote$ in y_2 is a read-modify-write.
  That is, y_2 can be written $y_3; r[j].vote = 0; r[j].vote := v; y_4$, where where $r[j].vote := 0$ is the memory fault, $r[j].vote = 0; r[j].vote := v$ is the read-modify-write by some processor p, and y_3 contains no explicit reference to $r[j].vote$. Note that the read-modify-write operations to the *vote* fields only change the value when it is non-zero. Then $y_1; r[j].vote := v; y_3; r[j].vote = v; y_4$ is a run of the algorithm that is similar to y, contains no more memory faults than y, and has one fewer (faulty) assignments of 0 to a *vote* field.

In each case the number of faulty assignments of 0 to a *vote* field decreases by one. The lemma follows by induction. ∎

Lemma 5.3 *Any complete run has a similar run, with no more memory faults, in which no memory fault occurs at $r[j].vote$ when the value is 0.*

Proof: Let y be a complete run of the form $y_1; r[j].vote := v; y_2$, where $r[j].vote := v$ is a memory fault and the value of $r[j].vote$ after y_1 is 0. Moreover, let this be the first such memory fault in y. By the previous lemma, it suffices to assume that no memory fault assigns 0 to any *vote* field in y. Since no processor ever writes 0 to any *vote* field, it follows that the value of $r[j].vote$ is 0 throughout y_1. There are several case:

- Either $v = 0$ or no operation in y_2 references $r[j].vote$.
  Then $y_1; y_2$ is a complete run that is similar to y, has one fewer memory faults than y, and has one fewer (faulty) assignments to $r[j].vote$ when the value is 0.

- $v \neq 0$ and the first reference to $r[j].vote$ in y_2 is a memory fault. Then y_2 can be written as $y_3; r[j].vote = v'; y_4$, where y_3 contains no reference to $r[j].vote$ and $v' \in \{-1, 0, 1\}$. Then $y_1; y_3; r[j].vote := v'; y_4$ is a run of the algorithm that is similar to y, has one fewer memory faults than y and the same number of (faulty) assignments to $r[j].vote$ when the value is 0.

- $v \neq 0$ and the first reference to $r[j].vote$ in y_2 is a read-modify-write.
  That is, y_2 can be written $y_3; r[j].vote = v; y_4$, where $r[j].vote = v$ is the read-modify-write by some processor p, and y_3 contains no explicit reference to $r[j].vote$. Since the value of $r[j].vote$ is 0 throughout y_1, this read-modify-write operation is from line 6 in the code, and the value returned is discarded by the executing processor. Hence, $y_1; y_3; r[j].vote = 0; r[j].vote := v'; r[j].vote := v; y_4$ is a run of the algorithm that is similar to y, has the same number of memory faults as y, and one fewer (faulty) assignments to $r[j].vote$ when the value is 0.

In each case the total number of memory faults is either reduced, or the total number remains the same, with one fewer (faulty) assignments to $r[j].vote$ when the value is 0. The lemma follows by induction. ∎

Lemma 5.4 *Any complete run has a similar run, with no more memory faults, in which memory faults to a vote field either changes its value from +1 to −1 or vice-versa.*

Proof: By the previous two lemmas, it suffices to consider complete runs in which no memory fault assigns 0 to a *vote* field, or over-writes a 0 in a *vote* field. The only remaining alternatives are memory faults which write +1 or −1, but do not change the value. These faults can be trivially deleted, resulting in a run satisfying the conditions of the lemma. ∎

Call runs satisfying the conditions of this lemma *legal runs*. Call read-modify-write operations to *vote* fields that actually change the value *successful read-modify-writes*. Call the $i-1$ reads by p immediately preceding a successful read-modify-write to $r[i].vote$ by p in line 6, the *collect* for that write.

Lemma 5.5 *In any legal run x, there are exactly $2m+1$ successful read-modify-writes, one to each vote field. Furthermore, the collects for any two such successful read-modify-writes are not concurrent: if $i < j$, the collect for the successful read-modify-write to $r[i].vote$ precedes the successful read-modify-write to $r[i].vote$, which in turn precedes the collect for the successful write to $r[j].vote$.*

Proof: By definition, every legal run is complete and the only memory faults to *vote* fields change the value from +1 to −1, or −1 to +1. In complete runs, every processor executes a read-modify-write on each *vote* field, so each is changed from 0 to 1 at least once. Once set, being non-zero is stable, so each *vote* field has exactly one successful read-modify-write.

The condition on collects holds trivially if both successful read-modify-writes and their collects are by the same processor. Suppose the read-modify-writes are by different processor, p and q, to $r[i].vote$ and $r[j].vote$, respectively. Note that q does an unsuccessful read-modify-write to $r[i].vote$ before the collect for $r[j].vote$ begins. Hence, the successful read-modify-write to $r[i].vote$ by p precedes this. The condition follows. ∎

Let s_k be a state of the system in a legal run of k atomic operations, and let AS_k be the remaining unexecuted faults to *vote* fields in a run i.e., m minus the number of such faults so far. Let CS_k be the number of 0's in the registers in s_k, define Σ_k to be the sum of the *vote* fields in s_k, $\Sigma_{i=1}^{2m+1} r[i].vote$, and finally, define Δ_k to be $|\Sigma_k| + CS_k - 2AS_k$.

Lemma 5.6 *For any $k \geq 0$, $\Delta_k > 0$.*

Proof: We first characterize the changes to these parameters that can result from any single step of the algorithm. That is, let π be a step in a legal run that changes the state from s_k to s_{k+1}.

1. π is a step of the adversary.
   That is, π is $r[i].vote := v$, where the value of $r[i].vote$ is $-v$ in s_k. Note that $CS_{k+1} = CS_k$ and $AS_{k+1} = AS_k - 1$. Here there are four key sub-cases.
   (a) $\Sigma_k = 0$. Then $|\Sigma_{k+1}| = 2$, and $\Delta_{k+1} = \Delta_k + 4$.
   (b) $0 < |\Sigma_k| < |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| + 2$, and again $\Delta_{k+1} = \Delta_k + 4$.
   (c) $0 < |\Sigma_k| = |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| = 1$, and $\Delta_{k+1} = \Delta_k + 2$.
   (d) $|\Sigma_{k+1}| < |\Sigma_k|$. Then $|\Sigma_{k+1}| = |\Sigma_k| - 2$, and $\Delta_{k+1} = \Delta_k$.
2. π is a successful read-modify-write.
   That is, π is $r[i].vote = 0; r[i].vote := v$. Then $CS_{k+1} = CS_k - 1$, $AS_{k+1} = AS_k$, and $\Sigma_{k+1} = \Sigma_k + v$. There are two subcases:
   (a) $|\Sigma_k| < |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| + 1$, and $\Delta_{k+1} = \Delta_k$.
   (b) $|\Sigma_{k+1}| < |\Sigma_k|$. Then $|\Sigma_{k+1}| = |\Sigma_k| - 1$, and $\Delta_{k+1} = \Delta_k - 2$.
3. π is any other atomic step.
   Then $CS_{k+1} = CS_k$, $AS_{k+1} = AS_k$, and $|\Sigma_{k+1}| = |\Sigma_k|$. Hence $\Delta_k = \Delta_{k+1}$.

In every (sub)case but one, 2b, the value Δ_{k+1} is greater than or equal to Δ_k. The problematic case is the occurance, then, of read-modify-write operations that decrease the value of $|\Sigma|$. Intuitively, such operations occur because faults have occurred so as to cause a processor to inadvertently "move" the value of $|\Sigma|$ in the wrong direction. As the inductive proof below shows, for each such read-modify-write operation that decreases $|\Sigma|$ by 2, there must be an earlier matching fault of type 1a, 1b or 1c that increases $|\Sigma|$ by at least 2, and the invariant follows.

The proof of the lemma proceeds by induction on the prefixes of the run. Clearly the invariant holds for the empty run ($AS_0 = m$; $CS_0 = 2m+1$; and $|\Sigma_0| = 0$). Let $\alpha\pi$ be a prefix of the run, where π is the $k+1$'st atomic operation and the invariant holds for every state $s_0, ..., s_k$. By the analysis above, no atomic step of the algorithm can falsify the invariant unless case 2b applies. In this case, π is a successful read-modify-write by some processor p, $r[i].vote = 0; r[i].vote := v$, and $|\Sigma_{k+1}| = |\Sigma_k| - 1$. Note that $\Sigma_k \neq 0$.

Since $\Sigma_k \neq 0$ and this is a legal run, in which the *vote* fields are first written sequentially by read-modify-write operations, $i > 1$. Moreover, by Lemma 5.5 $\alpha = \alpha_1$; $r[i-1].vote = 0$; $r[i-1] := v'$; $\alpha_2\pi$, where $r[i-1].vote = 0$; $r[i-1] := v'$ is the successful read-modify-write to $r[i-1].vote$ and there are no successful writes in α_2. Let s_j be the state at the beginning of α_2, just after the successful read-modify-write to

$r[i-1].vote$. By induction, $\Delta_j > 0$. Also by Lemma 5.5, α_2 contains the $i-1$ reads in the collect by p that precedes π. In addition, since none of the operations in α_2 are successful read-modify-writes, by the analysis above $\Delta_j \leq ... \leq \Delta_k$.

Next, consider the sequence of values $\Sigma_j, ...\Sigma_k$. We examine cases depending on the sign of the sum collected by p, and show that a fault described in case 1b or 1c must occur in α_2, implying $\Delta_k \geq 3$, and so $\Delta_{k+1} \geq 1$.

- The collect sums to a valid value. There are two subcases.
  - Some Σ in $\Sigma_j, ...\Sigma_k$ has the same sign as the sum collected.
    Since $|\Sigma_{k+1}| = |\Sigma_k| - 1$, the sign of the collect and hence of Σ is different than the sign of Σ_k. This must be due to a fault of type 1b or 1c, above. That is, there exist Σ_r and Σ_{r+1} in this sequence such that either $\Sigma_r = 0$ and $|\Sigma_{r+1}| = 2$, or $|\Sigma_r| = |\Sigma_{r+1}| = 1$ and $\Sigma_r = -\Sigma_{r+1}$. Then either $\Delta_{r+1} = \Delta_r + 4 \geq 5$ or $\Delta_{r+1} = \Delta_r + 2 \geq 3$, respectively, so $\Delta_k \geq 3$. Hence, $\Delta_{k+1} \geq 1$.
  - No Σ in $\Sigma_j, ...\Sigma_k$ has the same sign as the sum collected.
    Then a majority of the $i-1$ registers each have the same sign as the collect and as v at some point in the interval, but not at the end of the interval. Then a fault in the interval must change the value of one of these, from v to $-v$. That is, there exist Σ_r and Σ_{r+1} in this sequence such that $|\Sigma_{r+1}| = |\Sigma_r| + 2$, and $\Delta_{r+1} = \Delta_r + 4 \geq 5$. Hence, $\Delta_{k+1} \geq 3$.
- The collect sums to 0. There are two subcases.
  - Some Σ in $\Sigma_j, ...\Sigma_k$ has value 0.
    Since $\Sigma_k \neq 0$, some fault must move the sum from 0. That is, there exist Σ_r and Σ_{r+1} in this sequence such $\Sigma_r = 0$ and $|\Sigma_{r+1}| = 2$. Hence, $\Delta_{r+1} = \Delta_r + 4 \geq 5$ and $\Delta_{k+1} \geq 3$.
  - No Σ in $\Sigma_j, ...\Sigma_k$ has value 0.
    That is, half the registers are read as positive, and half as negative. Suppose first that there exist Σ_r and Σ_{r+1} in the sequence $\Sigma_j, ...\Sigma_k$ that have different sign: that $|\Sigma_r| = |\Sigma_{r+1}| = 1$ and $\Sigma_r = -\Sigma_{r+1}$. Then $\Delta_{r+1} = \Delta_r + 2 \geq 3$ and $\Delta_k \geq 3$. Hence, $\Delta_{k+1} \geq 1$.
    Suppose next that all of $\Sigma_j, ...\Sigma_k$ have the same sign. Since $|\Sigma_{k+1}| = |\Sigma_k + v| < |\Sigma_k|$, they have different sign than v. Since the collect read half the registers with $vote = -v$ and half with $vote = v$. Since the Σ all have sign different than v, some fault changes a value from v to $-v$ in α_2. That is, there exist Σ_r and Σ_{r+1} in the sequence such that $|\Sigma_{r+1}| = |\Sigma_r| + 2$, and $\Delta_{r+1} = \Delta_r + 4 \geq 4 \geq 5$. Hence, $\Delta_{k+1} \geq 3$.
- The collect is nonzero and invalid.
  By Lemma 5.1, all $i-1$ of the earlier successful read-modify-writes wrote the only valid value, v, yet the summed collect had opposite sign. Hence, in α at least $\lceil \frac{i+1}{2} \rceil$ registers had faults changing the value from v to $-v$. Recall that AS_{k+1} is m minus the number of faults in $\alpha\pi$; hence, $AS_{k+1} \leq m - \lceil \frac{i+1}{2} \rceil$, and $2AS_{k+1} \leq 2m - i$. Hence, we have

$$\begin{aligned} \Delta_{k+1} &= |\Sigma_{k+1}| + CS_{k+1} - 2AS_{k+1} \\ &= |\Sigma_{k+1}| + (2m+1-i) - 2AS_{k+1} \\ &\geq |\Sigma_{k+1}| + 2m + 1 - i - (2m-i) \\ &\geq |\Sigma_{k+1}| + 1 \\ &\geq 1 \end{aligned}$$

∎

Lemma 5.7 (Agreement) *All processors decide on the same value.*

Proof: Consider Σ_k in the system state s_k, immediately after the the last register, $r[2m+1]$, has been written. By Lemma 5.6, $2AS_k < |\Sigma_k|$. Henceforth, there are insufficient faults remaining to change the sign of Σ_k, or reduce it to 0. Since all the reads in any final collect (upon which any decision is based) are made after s_k, all processors decide on the same value. ∎

5.2 Unbounded failures per register

The protocol of Figure 5 does not work when the number of faults per register is unbounded. For the case of ∞-faulty registers we use a slightly different technique.

Consider an array of $2i+1$ read-modify-write registers over the range $\{-1, 0, +1\}$, initialized to 0. Each processor scans the registers, one at a time, writing an input value $d \in \{-1, +1\}$ to each register whose value is zero and returning the resulting non-zero value of the register. After scanning all $2i+1$ registers, the processor outputs the majority of the values returned from the registers. Call this construct *filter*(i). Three observations are important:

1. If at most i registers are faulty then each processor sees a majority, which is the input value for some processor.
2. If at most i registers are faulty and all processors have the same input value then all processors see the same value in majority.
3. If there are no memory faults then all processors compute the same majority value, which is the input of some processor.

```
Protocol for processor p, v_i ∈ {-1,+1}:

type   reg = record minus, plus: in {0,1}
                    val: in {-1,0,+1}
shared r: array[1..(f+1)^2] of reg, initially (0,0,0)
local v_o: in {-1,0,+1}, initially 0

% indicate that v_i is a valid input
for j = 1 to 2f+1 do
  if v_i = +1 then RMW(j, plus, 1)
  else RMW(j, minus, 1)
v_o := decide(f, v_i) % make final decision

function decide(i, input)
if i = 0 then return(RMW(1, val, input))
  else input := valid(decide(i - 1, input))
    sum := 0
    for j = 1 to 2i + 1 do
      sum := sum + RMW(i^2 + j, val, input)
return(sign(sum))
end_function

functions valid and RMW are as in Figure 5
```

Figure 6: Consensus in the presence of f, ∞-faulty RMW registers: $(f+1)^2$ constant-size registers.

5.2.1 RMW registers of constant size

Simply having each processor use $f+1$ copies of *filter*(f), (or $(f+1)(2f+1)$ 3-valued RMW registers) with v_i as the input to the first copy and the output of the previous *filter* as the input to the next results in a consensus on the final output. However, the protocol in Figure 6 employs a recursive construction and the validation technique from the algorithm in Figure 5 to achieve consensus, while using about half as many registers (each register is slightly larger, requiring two boolean fields and one three-valued field).

Theorem 13 *For any $f \geq 0$, there is a strongly-wait-free consensus algorithm using $(f+1)^2$ 12-valued read-modify-write registers, at most f of which are ∞-faulty:*

$$\mathbf{CONS}_\infty(\mathit{constant\_rmw}, f, \mathit{consensus}) \leq (f+1)^2.$$

Sketch of proof: The algorithm is recursive; assume a strongly-wait-free consensus algorithm for $i-1$ ∞-faulty registers, called *decide*$(i-1, input)$. With *filter*(i), strongly-wait-free consensus with i ∞-faulty registers can be solved recursively as follows: first, as in the algorithm of Figure 5, each processor p *validates* its input value by setting bits in $2f+1$ registers. Then it runs *decide*$(i-1, input_p)$, and obtains a decision value, d. If p finds that d is valid (bits in at least $f+1$ corresponding validation fields are set), p enters the filter with d as input, otherwise p enters the filter with $input_p$. The output of the filter is then the final decision value.

1. If fewer than i faulty registers occur in *decide*$(i-1)$, then all processors leave *decide*$(i-1, input_p)$ with the same valid value, d, and by the second observation, all leave *filter*(i) with the value d.
2. If however, there are i faulty registers in *decide*$(i-1, input_p)$, then processors leave *decide*$(i-1, input_p)$ with arbitrary values, but the validity check ensures they enter *filter*(i) with a value which is the input of *some* processor. Since in this case there are no faults in *filter*(i), by the third observation, all processors leave *filter*(i) with the same value d, and d is the input of some processor.

In either case, the validity and agreement conditions are satisfied by *decide*$(i, input_p)$. Since both *decide*$(i-1, input)$ and *filter*(i) are strongly-wait-free, so is *decide*$(i, input_p)$.

Figure 6 presents a strongly-wait-free consensus algorithm implementing this recursion. As in the algorithm of Figure 5, only one set of validity bits is needed, as any processor that reads $f+1$ bits for value d, in any level of the recursion, sets all $2f+1$ bits before proceeding. This has the consequence that no value is written by a processor unless it has first set all $2f+1$ validity bits for that value. Hence, at any level i of the recursion, a processor can compute an invalid majority only if $i+1$ faults have occurred. The full proof of this algorithm is left to the full paper. The total number of registers used is $\sum_{i=0}^{f} 2i+1 = (f+1)^2$. ∎

5.2.2 RMW registers of exponential size

Another simple consensus algorithm can be designed using *filter*(f) and $3f+1$ registers, each of exponential size.

Theorem 14 *For any $f \geq 0$, there is a strongly-wait-free consensus algorithm using $3f+1$ read-modify-write registers of exponential size, at most f of which are ∞-faulty:*

$$\mathbf{CONS}_\infty(\mathit{exponential\_rmw}, f, \mathit{consensus}) \leq 3f \quad 1.$$

Proof: The algorithm iteratively runs *filter*(f) over (a specific enumeration) of every subset of the registers of size $2f+1$, using a different bit-field in each register in each run of *filter*(f), and using the output from the previous *filter* as input to the next. Figure 7 presents the details of the algorithm.

The algorithm requires exponential time for each processor, but is still strongly-wait-free. Since f bounds the total number of register faults, the majority computed by each processor is always the input of some processor, and the validity condition is satisfied. Moreover, since at least one instance of *filter* uses no faulty registers, the third observation above implies that at some point all processors exit *filter* with the same value. By the second observation, all processors exit each later *filter* with that value. The agreement condition follows. ∎

Protocol for processor p, $v_i \in \{-1,+1\}$:

type $\mathcal{S}$ = subsets of $\{1,...,3f+1\}$ of size $2f+1$
$val = \{-1,0,+1\}$
$register = reg[S]$: **array** $S \in \mathcal{S}$ of val
% Each reg. is an array of $\binom{3f+1}{2f+1}$ val fields.
shared $r[j]$: **array** $1 \le j \le 3f+1$ of $register$
% initially all val fields 0
local v_f: ranges over $\{-1,0,+1\}$, initially 0

$input := v_i$
for each subset $S \in \mathcal{S}$, in enumeration order **do**
  $sum := 0$
  for each $j \in S$ **do**
    $sum := sum + RMW(r[j],[S],input)$
  $input := sign(sum)$
$v_f := input$

Figure 7: Consensus in the presence of f, ∞-faulty RMW registers: $3f+1$ exponential-sized registers.

5.3 Faulty RMW registers are universal

Herlihy has defined the notion of a *universal* object, as an object that can be used to construct a wait-free implementation of any other object [Her91]. He then showed that consensus for n processes is universal for systems with at most n processes.

Theorem 15 *Read-modify-write registers are universal objects for systems with at most n processes, if a bounded number of them are ∞-faulty: For all objects X, $\mathbf{CONS}_\infty(unbounded\_size\_rmw, f, X) < \infty$.*

Proof: The proof relies on Herlihy's construction in his proof of the universality of consensus, which uses $O(n^3)$ reliable atomic read/write registers of unbounded size and $O(n^3)$ reliable consensus objects over a bounded domain. The data-oblivious constructions of Theorem 5 allow us to construct the reliable unbounded atomic read/write registers from ∞-faulty read-modify-write registers.

Next, by Theorem 13, we know that sufficiently many read-modify-write registers can be used to implement n-processor binary consensus, when any fixed number of them may be ∞-faulty. Known constructions can be used to implement multi-valued consensus from binary [TC84]. Moreover, the read-modify-write objects used in the binary construction and so in the multi-valued constructions may be easily reset as Herlihy's universal construction requires. (Specifically, the reset operations are not concurrent with other consensus operations.) ∎

6 Discussion

We have studied memory failures that are restricted in total number, or in the number of data objects that may be affected. Memory failures may be restricted in time, as well. For example, such constrained memory faults are studied in work on *self-stabilizing* systems defined by Dijkstra [Dij74]. Self-stabilizing systems are required to recover once the final memory fault occurs, and the system is in an arbitrary state. Hence, such failures may affect local memories of processors, as well as the shared memory. However, work in self-stabilization (necessarily) considers only non-terminating control problems such as the mutual exclusion problem, whereas we also study short-lived objects such as the consensus problem, in which a processor makes a single, irrevocable decision after a finite number of steps.

Three previous papers investigated *initialization failures* that are restricted to the shared memory, and inspired our work [FMRT90, FMT91, MTY92]. A shared register is subject to initialization failure if the shared register contains an arbitrary unknown value when the algorithm begins. These three papers assume that all the shared registers are subject to initialization failures, and study both control and decision problems.

In this paper we used the consensus problem to explore properties of faulty shared memory. Much is known about the consensus problem in other models. See, e.g. [Abr88, Fis83, FLM86, FLP85, LA87]. We also investigated the question of constructing reliable registers in an unreliable environment. This relates to the problem of implementing one type of shared objects from another. Such work includes: [Lam86, VA86, Blo87, BP87, SAG87, LTV89, Her91].

6.1 Open problems

There remain many unresolved issues related to shared memory failures in distributed systems. Faulty versions of other shared data objects, such as multi-valued test-and-set registers, m-registers, or compare-and-swap, are of interest. We have tight bounds on only a few problems; more efficient constructions and corresponding lower bounds would also be interesting. For example, our implementations of consensus from read-modify-write objects suggest a trade-off between the number of read-modify-write objects and their size–we conjecture that $2f+1$ registers are sufficient, but whether they must be large is not clear.

It would be particularly interesting to implement memory-fault tolerant data objects directly from similar, faulty objects, such as test-and-set from test-and-set, without using atomic registers, or read-modify-write from read-modify-write, without using an unbounded universal construction.

Theorem 3 describes a composition in which faulty

objects are first used to construct fault-free objects, which can then be used to construct other fault-free objects. Our fault definition is not general enough to support the dual result: using fault-intolerant constructions from faulty objects, then using the resulting faulty objects in a fault-tolerant construction. The problem is to characterize the result of applying a fault-intolerant construction of a type Y object from faulty objects of type X. We are currently considering general definitions that would support such a result, and more complex compositions of two or more fault-tolerant compositions.

All our solutions are deterministic. It would be interesting to explore the use of randomization to tolerate memory failures. Also, there is much work to be done in exploring the effect of memory failures in other models, such as synchronous or semi-synchronous models.

References

[AAD+90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1–13, August 1990. Submitted for publication.

[Abr88] K. Abrahamson. On achieving consensus using shared memory. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 291–302, 1988.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 249–259, 1987.

[BP87] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 222–231, 1987.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 86–97, 1987.

[Dij74] E. W. Dijkstra. Self-stablizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Foundations of Computation Theory*, 127–140. Lecture Notes in Computer Science, vol. 158, Springer-Verlag, 1983.

[FLM86] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[FMRT90] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. In *Proc. 22st ACM Symp. on Theory of Computing*, 106–116, May 1990. See also Technion Report #644, Computer Science Department, The Technion, August 1990.

[FMT91] M.J. Fischer, S. Moran, and G. Taubenfeld. Space-efficient asynchronous consensus without shared memory initialization. Submitted for publication, 1991.

[Her91] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Lam86] L. Lamport. On interprocessor communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[LA87] C. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processors. *Advances in Computing Research*, 4:163–183, 1987.

[LTV89] M. Li, J. Tromp, and P. M.B. Vitanyi. How to share concurrent wait-free variables. In *ICALP*, 1989. Expanded version: Report CS-R8916, CWI, Amsterdam, April 1989.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[MTY92] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[NW87] Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 232–248, August 1987.

[Pet83] G. L. Peterson. Concurrent reading while writing. *Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[PF77] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. 9th ACM Symp. on Theory of Computing*, 91–97, 1977.

[PB87] G. L. Peterson and Burns J. E. Concurrent reading while writing II : The multi-writer case. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, 383–392, 1987.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus.* PhD thesis, M.I.T., August 1988. Ph.D. Thesis.

[Rab82] M. O. Rabin. N-processor mutual exclusion with bounded waiting by $4 \log n$ shared variables. *Journal of Computer and Systems Sciences*, 25:66–75, 1982.

[SAG87] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register revisited. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 206–221, 1987.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14:473–540, 1982.

[Tro89] J. Tromp. How to construct an atomic variable. In *3rd International Workshop on Distributed Algorithms*, 1989. Lecture Notes in Computer Science, vol. 392 (eds.: J.C. Bermond and M. Raynal), Springer-Verlag 1989, 292–302.

[TC84] R. Turpin and B. Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984.

[VA86] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 223–243, 1986. Errata, Ibid., 1987.

Concurrent Counting

(EXTENDED ABSTRACT)

Shlomo Moran* Gadi Taubenfeld† Irit Yadin*

Abstract

Our purpose is to implement clocks and, in general, counters in a shared memory environment. A concurrent counter is a counter that can be incremented and read, possibly at the same time by many processes. We study counters that achieve high level of concurrency and thus are likely to reduce memory contention; require only weak atomicity and thus are easy to implement; do not depend on the initial state of the memory and hence are more robust to memory changes; and are wait-free – one process cannot prevent another process from finishing its increment or read operations – and thus can tolerate any number of process failures. We concentrate on providing upper and lower bounds on the space complexity of the counters studied.

1 Introduction

1.1 The Concurrent Counter Problem

Counters are basic objects which are used in various computer applications. A counter (mod m) holds an integer from $0, .., m-1$, and enables two basic operations: `increment` – which increments the value by one (mod m), and `look` – which gets the current value. A *concurrent* counter is a counter in a shared memory environment, which can be *incremented* and *looked*, possibly at the same time, by many processes. Throughout the paper we assume that a counter is always incremented modulo m for some fixed m.

Since counters, or objects which include counters as a special case, are involved in many protocols, results concerning concurrent counters are likely to have implications on other problems in asynchronous computations. The implementation of concurrent counters raises many basic problems concerning the possibility and the cost of multi-process coordination in an asynchronous shared memory systems. In this paper we study two types of concurrent counters.

A *static counter* guarantees that the counter will hold the correct value even if it is incremented and read concurrently by several processes. However, it only guarantees that a `look` operation returns the correct value when it is not concurrent with any `increment` operation. In the case that a `look` operation overlaps an `increment` operation, a static counter may return an arbitrary value.

A *dynamic counter* guarantees that the counter will hold the correct value even if it is incremented concurrently by several processes and that processes can read a correct value of the counter even if the read is concurrent with other increments or reads. That is, for a given `look` operation, let c_1 be the initial value of the counter plus the number of `increment` operations that were completed before the look operation started, and let c_2 be the initial value of the counter plus the total number of `increment` operations that were initiated before the look operation was completed. Then the `look` operation should return some value between c_1 and c_2.

The most common example of a concurrent counter is probably a global clock, which can be incremented by one process, but arbitrarily many

*Computer Science Department, Technion, Haifa 32000, Israel.

†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

PoDC '92-8/92/B.C.

© 1992 ACM 0-89791-496-1/92/0008/0059...$1.50

processes may gets its value [Lam90]. In implementing a global clock one would not like to set a bound on the number of processes that may see the time in this clock, though in general only one process is allowed to change its value.

Another example of a possible use of a concurrent counter is in protocols for the wakeup problem [FMRT90]. In some of the protocols for this problem every process starts its participation in the protocol by incrementing a counter. The counters used in the wakeup protocols differ from the one used for global clocks in two important features: first, it can be incremented by many distinct processes; second, in each run of the protocol, the counter of the wakeup protocols can be incremented at most some bounded and known number of times. One more example is in fault-tolerant solutions for the consensus or leader election problems [Fis83, KMZ84, Pet82]. In such solutions it is sometimes required to count the number of processes that already voted. For example, in [FMT91] a consensus is reach after some process observes that a counter is incremented by more than half of the processes.[1]

It is easy to implement a concurrent counter (mod m) using one register of size $\log m$ bits, whose values are $\{0, ..., m-1\}$. To increment or read the counter a process first *locks* the access to the counter, modifies or gets its value, and then *unlocks* the access to the counter. This approach ensures the correctness of the implementation but causes bottlenecks when there is high memory contention since all `increment` and `look` operations must be serialized. Moreover, it is necessary to implement a lock operation on a large – $\log m$ bits – register, and in the presence of failures access to the counter may get blocked.

Our goal is to design solutions to the concurrent counter problem that achieve a high level of concurrency, and thus are also likely too reduce memory contention; require only weak atomicity and thus are easy to implement; do not depend on the initial state of the memory and hence are more robust to memory changes; and are wait-free – one process cannot prevent another process from finishing its `increment` or `look` operations – and thus can tolerate any number of process failures. Finally, and most important, we want our solutions to use as little shared space as possible.

[1]The model in [FMT91, FMRT90] assumes only one shared register. Some of the solutions can use our implementation of dynamic counters.

1.2 Computational Model

The model consists of a fully asynchronous collection of identical anonymous deterministic processes that communicate via bounded size shared registers which are initially in an arbitrary unknown state. The registers that are used are ***binary*** registers unless otherwise indicated. Also, unless otherwise stated, it is assumed that access to a shared register is via atomic "read-modify-write" instructions, which, in a single indivisible step, reads the value of the register and then writes a new value that can depend on the value just read.

We note that implementing a read-modify-write on a single bit can be done in one time unit by a simple, one-input one-output, logical gate, and is in general much easier than on a register of larger size.

In some cases we develop protocols under the assumption that only one process can increment the counter, and in such cases we assume only read/write atomicity. That is, in one atomic instruction a process may either read from or write to a register. When two or more identical processes may increment the counter, the read-modify-write atomicity cannot be replaced by the weaker read/write atomicity.

We assume that any number of processes can fail. The only kind of failures we consider are crash failures, in which a process may become faulty at any time during its execution, and when it fails, it simply stops participating in the protocol.

Assuming an arbitrary unknown initial state relates to the notion of self-stabilizing systems defined by Dijkstra [Dij74]. However, Dijkstra considers only non-terminating control problems such as the mutual exclusion problem, whereas our implementations of counters can also be used to solve decision problems such as the wakeup and consensus problems, in which a process makes an irrevocable decision after a finite number of steps.

It seems that this model more accurately reflects reality, where in many cases all processes are programmed alike, there is no global synchronization, and it is not possible to simultaneously reset all

parts of the system to a known initial state. Our model is similar to the shared memory model studied in [FMRT90], except that in [FMRT90], it is assumed that there is a *single* finite sized shared register and that access to the shared register is via atomic read-modify-write instruction.

1.3 Summary of Results

We present lower and upper bounds on the number of shared bits required to implement static and dynamic counters as a function of the number of processes that are allowed to increment the counter. In some cases we present implementations that are very efficient in both space and time, while in other cases we show that any implementation must be very inefficient. All the results are stated under the assumption that the basic atomic operations are performed on single binary registers (bits). These results can be improved (in terms of the space used) when larger registers are available (for details see later). Since the counter is a collection of bits which can be tested separately, we will assume in our upper bounds that m is a power of 2. Finally, the notion of a static (dynamic) n-counter protocol means that no more than n processes are allowed to increment the counter. We may let n be infinity (∞) which means that there is no restriction on the number of processes that can increment the counter. We will also use the notion of an *ℓ-bounded* counter protocol which is a protocol where a total of at most ℓ increments are performed in any of its runs. We give a brief overview of our results below.

OPTIMAL STATIC ∞-COUNTERS: We present a static ∞-counter protocol that uses only $\log m$ bits. That is, in the protocol there is no bound on the number of processes that are allowed to increment the counter and it matches the trivial $\log m$ bits lower bound.

OPTIMAL DYNAMIC 1-COUNTERS: In the case when only one process is allowed to increment the counter, we are able to construct a protocol that uses only $\log m$ bits, and hence matches the trivial lower bound. Thus, our protocol gives an optimal solution to Lamport's global clock problem. In designing the protocol we use, in a new way, reflected binary Gray code. Consequently, incrementing the counter requires a single write operation, and hence, unlike other implementations studied in the literature, it consists of a single atomic step in any model that supports read/write atomicity on single bits.

DYNAMIC COUNTERS: We present an m-bounded dynamic ∞-counter protocol which uses exactly m bits. Then we use it to construct an (unbounded) dynamic n-counter which uses $\alpha(\log m + 1)$ shared registers, where α is the smallest power of 2 that is not smaller than n.

LOWER BOUND: Let $k = \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$. We prove that any ℓ-bounded dynamic n-counter protocol must use at least k registers. This result holds even when the processes have unique identifiers and there is only one possible initial state. Furthermore, by making various restrictions on the way processes may increment the counter we are able to tighten these bounds.

1.4 Related Work

In [Lam90], Lamport developed algorithms to implement both monotonic and cyclic multiple-word clocks that are updated by one process and read by one or more processes. Lamport's cyclic clock problem is a special case of the concurrent (dynamic) counter problem where there is only one process that can increment the counter (i.e., $p = 1$). His solution uses $2\log m + 2$ registers, and relies on the assumption of a known initial value.

Aspnes, Herlihy and Shavit [AHS91] introduced a fundamental new class of networks, called *counting networks*. They used counting networks to construct various objects such as a *shared counter* which is an object that can issue the numbers 0 to $m - 1$ in response to m requests by processes. Counting networks can be viewed as objects which support one atomic operation, which consists of both `increment` and `look` . It seems that counting networks cannot support a `look` operation (without incrementing the counter). The two constructions of counting networks in [AHS91] require $O(m \log^2 m)$ binary registers. Our problem seems to be related to but different from counting networks: (1) As mentioned above, it is not clear whether counting networks can support a `look` operation without incrementing the counter, while we implement a `look` operation in our solutions. (2) All implementations of counting networks rely on

the assumption of a single initial state, that is, all shared registers require initialization, while we require that a solution to the concurrent counter problem will work for any possible initial state.

In [FMRT90], Fischer, Moran, Rudich and Taubenfeld investigated a deceptively simple problem called the wakeup problem. The goal is to design a protocol for n asynchronous identical processes in a shared memory environment such that at least one process eventually learns that at least τ processes have waked up and begun participating in the protocol [FMRT90]. All the solutions for that problem, except one, use some implementation of a counter in order to count the wakeup processes. There is one solution however, the see-saw protocol, where the counting is done somehow in the local memory of the processes and it requires only two bits of shared memory which are used for communication. The see-saw protocol cannot tolerate even one faulty process, and it seems that the approach taken in designing it cannot be adopted to solve the concurrent counter problem.

As mentioned earlier, when two or more identical processes may increment the counter, the read-modify-write atomicity, assumed in this paper, cannot be replaced by the weaker read/write atomicity. However, it is shown in [AH90, AH90, AG91, Lam77], and follows from the algorithm in Subsection 4.1, that it is possible to implement a counter using only read/write atomicity, when the processes have unique identifiers. In these implementations the basic correctness condition is *linearizability*. That is, although operations of concurrent processes may overlap, it should provide the illusion that each counter operation is atomic, while preserving the order in which operations that do not overlap happen.

In the full paper we also cover the notion of a *serial counter*, which intuitively is a dynamic counter in which the executions of the `increment` and `look` operations are linearizable. The *static*, *dynamic*, and *serial* counters bear some similarity to the notions of *safe*, *regular* and *atomic* registers defined by Lamport in [Lam86]. In a *safe* register, it is assumed only that a read not concurrent with any writes obtains the correct value. A *regular* register is a safe register in which a read that overlaps a write obtains either the old or new value. An *atomic* register, is a safe register in which the reads and writes behave as if they occur in some linear order.

2 An Optimal Static ∞-Counter

In this section we present a static ∞-counter protocol that uses only $\log m$ bits. That is, in the protocol there is no bound on the number of processes that are allowed to increment the counter and it matches the trivial $\log m$ bits lower bound.

Theorem 2.1 *There is a static ∞-counter protocol that uses $\log m$ registers.*

In order to prove the theorem we describe the *Positional Protocol*. In this protocol a process may change the value of several registers during a single `increment` operation. In section 5, we show that any optimal static ∞-counter must allow processes to change the value of more than one register during a single `increment` operation, and that there is no dynamic ∞-counter protocol that achieves the same $\log m$ space complexity. We point out that the protocol is not even dynamic 1-counter protocol.

In the Positional Protocol the content of the $k = \log m$ shared registers $r_{k-1}, \cdots, r_0$ are viewed as a binary representation of the value of the counter. The `increment` operation is performed by the straightforward (sequential) algorithm for incrementing a binary number. That is: scan the the registers from right to left (starting with r_0); when scanning register r_i, do the following: (1) flip r_i, and (2) if the value of r_i was 1 before it was flipped and $i < k-1$, then repeat this operation on register r_{i+1}, else terminate the `increment` operation. The `look` operation is performed by simply reading the content of the registers.

The correctness proof of this simple implementation is somewhat complicated by the fact that several `increment` operations may take place simultaneously. The proof is based on showing that in any execution in which k complete `increment` operation are performed (for arbitrary k) and no other increment is initiated, the number of times each register is changed depends only on the initial content of the shared registers and on k, regardless the order by which the registers were accessed by the various processes.

The ideas can be generalized to work for an arbitrary m. Let us write m's prime factorization as $m = \prod_{i=1}^{t} v_i^{e_i}$ with $i \neq j$ implies $v_i \neq v_j$ and for all i, v_i is prime. Then the Positional Protocol can be easily modified to work modulo m, using e_i v_i-valued registers, for all $1 \leq i \leq t$. Also, using *residue number systems* we can get an even faster solution. For a given m we can look at the prime factorization of m, and then use powers of primes as small counters for constructing a counter mod m. The properties of residue number systems allow us to work on the smaller counters concurrently.

3 An Optimal Dynamic 1-Counter

In the previous section we presented a space optimal static ∞-counter protocol. The situation with dynamic counters is considerably more involved, and in the general case dynamic counters will require much more space than static ones. One specific case where we were able to design a dynamic counter that matches the trivial $\log m$ bits lower bound is the case where only one process can increment the counter.

Theorem 3.1 *There is a dynamic* 1*-counter protocol that uses* $\log m$ *shared registers.*

To prove the theorem we present a protocol, where each `increment` operation changes the value of exactly one bit, and a `look` operation never changes the value of a bit. The motivation for this protocol is due to a lemma in which we prove that in any dynamic 1-counter protocol that uses $\log m$ bits, a process must change the value of exactly one bit during a single `increment` operation, and no bit can be flipped during a `look` operation. We call such a protocol an *1-flip counter*. It is interesting to note that dynamic 1-flip counters are, in fact, serial counters, since we can order the `look` and `increment` operations of each execution in a complete order, which is consistent with the partial order defined by that execution, as follows:

1. First order the `increment` operations according to the order of the read-modify-write instruction that flipped the value of a register. Note that since this is a 1-flip protocol, this order is well defined.
2. Then, order each `look` operation which overlaps at least one `increment` operation, after one of the `increment` operations which overlap it, where the value of the counter immediately after that increment operation equals the value returned by the `look` operation.
   After this step, all the `increment` operations are ordered, and all the `look` operations are ordered relative to the `increment` operations.
3. Finally, complete the partial order of `look` operations that appear between two consecutive `increment` operations to a complete order in an arbitrary way.

3.1 Preliminaries

For many years, Gray code is used for counting [Gar72, Gil58, Gra53, Koh70]. Increments are done, by only one incrementor, according to the code, while a `look` operation simply reads the counter digits, and convert them to the appropriate number. This technique works *only* if a `look` operation is much faster than an `increment` operation. In the case where few `increment` operations are concurrent with a `look` operation, the look might return a wrong answer. In our framework we do not assume anything about the relative speeds of `increment` and `look` operations, hence the above naive use of Gray code does not solve our problem.

Reflected binary Gray code (abbv. Gray code) is a well known method to order all binary words of any given length k in a cyclic order, such that two successive words differ in exactly one bit. It is called reflected code because it can be generated by the following simple algorithm. Start with 0,1 as a one-digit Gray code, then reflect and append the digits to get 0,1,1,0. Next put 0's in front of the first two numbers and 1's in front of the last two numbers. The result is a two-digit gray code 00,01,11,10. To extend an i-digit Gray code to an $(i+1)$-digit code, reflect the the i-digit code and, as before, put 0's in front of the first half of these numbers and 1's in front of the last half. Note that the resulting code is cyclic in that the first and last numbers also differ at only one position. Let $G_k = (g_0, g_1, \cdots, g_{m-1})$ be all the binary words of length $k = \log m$ ordered by Gray code (where g_0 is the allzero word). Let *gray* be the 1-1 mapping defined by $gray(g_i) = i$. Our protocol represents

each integer $i \in \{0, \cdots, m-1\}$ by g_i.

An important known property is that the mapping *gray* and its inverse $gray^{-1}$ are computable by an on-line linear time algorithm. To convert a standard binary number to its reflected Gray equivalent start with the digit at the right and consider each digit it turn. If the next digit to the left is 0, let the former digit stand. If the next digit to the left is 1, change the former digit. The leftmost digit is assumed to have 0 on its left and therefore remains unchanged. To convert back again consider each digit in turn starting at the right. If the parity (sum) of all digits to the left is even, let the digit stay as it is. If the parity is odd, change the digit.

There is also a known on-line linear time algorithm to find the successor of a word in G_k. For each k-bits word $v = [v_{k-1} \cdots v_0]$, define the *critical index* of v to be the minimal index j in $\{0, \cdots, k-1\}$ such that the $k-j$ bits prefix of v, $[v_{k-1} \cdots v_j]$, has even parity, and if there is no such index (i.e., $v = [10 \cdots 0]$) then $j = k-1$. Then the successor of v in G_k is obtained from v by flipping the bit v_j, where j is the critical index of v.

In the full version we prove the following properties which are used in the construction of the protocol. For any word w where $|w| = i < k$, all the k-bits words whose prefix is w appear consecutively in G_k. Denote by $first(w)$ the first word in G_k that has w as a prefix. Similarly, denote by $last(w)$ the last word in G_k which has w as a prefix. Finally, $middle(w)$ is the the unique word v, for which $gray(v) = \frac{gray(first(w)) + gray(last(w))}{2}$. Let $par(w)$ be the parity w. Then $first(w) = w \cdot par(w) \cdot 0^{k-i-1}$ (i.e., w followed by the parity bit of w followed by $k-i-1$ zeroes), $last(w) = w \cdot \neg par(w) \cdot 0^{k-i-1}$, and $middle(w) = w \cdot par(w) \cdot 1 \cdot 0^{k-i-2}$.

3.2 The Gray code Counter

We can now give an informal description of the *Gray code protocol*. The code of the protocol is omitted from this abstract as well as its proof, which is rather involved. For the rest of this section m and $k = \log m$ are fixed, and k is the number of registers used for the counter.

The `increment` operation is performed by reading the content $v = [v_{k-1} \cdots v_0]$ of the counter registers, and flipping v_j, where j is the critical index of v described above. We notice that since there is only one process that may increment the counter, it has to read the counter only one time, at the beginning of the protocol.

The protocol for `look` uses a function called *4-way scan* whose purpose is to take a snapshot of the content of the counter registers, and to return a word g_i such that i is a valid value of the counter. This function is described next. It first reads the registers from right to left and gets a word $a = [a_{k-1} \cdots a_0]$. That is, a_0 is read first and a_{k-1} is read last. Then it reads the registers from left to right and gets a word $b = [b_{k-1} \cdots b_0]$ (this time b_{k-1} was read first). Then, again, it reads the registers from right to left and gets a word $c = [c_{k-1} \cdots c_0]$, and, finally it reads the registers from left to right and gets a word $d = [d_{k-1} \cdots d_0]$.

Let w be the maximal common prefix of the words a, b, c and d, and let $|w| = i$. In case $i < k$, let x_1, x_2, x_3 and x_4 be the $i+1$st bit of a, b, c and d respectively. (I.e., $w \cdot x_1$ is a prefix of a.) The function checks the following conditions sequentially:

1. if $a = b$ then return a;
2. elseif $c = d$ then return c;
3. elseif $|w| = 0$ then return $last([a_{k-1}])$;
4. elseif $x_1 = \neg par(w)$ and $x_2 = x_3 = x_4 = par(w)$ then return $first(w)$;
5. elseif $x_1 = x_2 = x_3 = \neg par(w)$ and $x_4 = par(w)$ then return $last(w)$;
6. elseif $x_1 = x_2 = \neg par(w)$ and $x_3 = x_4 = par(w)$ then return $last(w)$;
7. elseif $x_1 = x_3 = \neg par(w)$ and $x_2 = x_4 = par(w)$ then return $last(w)$.
8. elseif all the above conditions fail then return $middle(w)$.

Using the *4-way scan*, it is now easy to describe the implementations of the `look` operation. The `look` operation calls the *4-way scan* function that returns a k bit word, g. The output of the `look` operation is $gray(g)$.

4 Dynamic Counters for Many Processes

In this section we present two dynamic counters. The first works when the number of `increment`

operations is bounded; the other works when the number of processes is bounded and known, and the number of **increment** operations is unbounded.

4.1 The Cyclic-flip Counter

In this subsection we present an m-bounded dynamic ∞-counter protocol – that is, a protocol that works as long as no more than a total of m increments are performed. Counters of this type are used (implicitly) in wakeup and consensus protocols.

Theorem 4.1 *There is an m-bounded dynamic ∞-counter protocol that uses m registers.*

The protocol presented here, called the Cyclic-flip protocol, uses m registers, which is exponentially more than required by the dynamic 1-counter of the previous section. In the next section we show that an exponential gap in the number of registers is unavoidable, even if it is assumed that all the registered are initialized, and the processes are allowed to be distinct.

4.1.1 The Counting Method

Like the Positional and the Gray code counters, the Cyclic-flip counter is based on a mapping from binary words onto $\{0, 1, \cdots, m-1\}$, where m is a power of two. However, this time the domain of the mapping is the words of length m, and not of length $\log m$ as in the previous cases. The mapping, called *number*, is defined as follows:
Let w be a binary word of length m (where m is a power of 2), and let $par(w)$ be the parity of w. When $m > 1$, let $w = w_2 \cdot w_1$ where $|w_2| = |w|/2$. Then,

$$number(w) = \begin{cases} 0 & \text{if } |w| = 1; \\ 2 \cdot number(w_2) + par(w) & \text{otherwise.} \end{cases}$$

That
is, $number(w) = \sum_{i=0}^{\log m - 1} v(i) \cdot 2^i$, where $v(i) = par(w(m-1) \cdots w(m - 2^{\log m - i}))$.
($v(i)$ is the parity of the $\frac{m}{2^i}$ leftmost bits of w.)
Thus, for m=8, $number(10110001) = 2^2 + 2^1 = 6$, since 10, 1011 have parity 1, and 10110001 has parity 0.

Given a word w, the **increment** operation is done by finding a word w' such that $number(w') = number(w) + 1$. For this we use the function *next*, which we define below.

Let w be a binary word of length $m = 2^k$. When $m > 1$, let $w = w_2 \cdot w_1$ where $|w_1| = |w_2| = |w|/2$. Then,

$$next(w) = \begin{cases} [0] & \text{if } w = [1]; \\ [1] & \text{if } w = [0]; \\ w_2 \cdot next(w_1) & \text{if } par(w_1) = par(w_2); \\ next(w_2) \cdot w_1 & \text{if } par(w_1) \neq par(w_2). \end{cases}$$

We define $next^\ell(w)$ recursively as follows: $next^0(w) = w$ and $next^\ell(w) = next(next^{\ell-1}(w))$, for $\ell > 0$.

Lemma 4.1 *Let w be a binary word of length m. Then, $next^\ell(w)$ and w differ by exactly ℓ bits, for any $0 \leq l \leq m$.*

Lemma 4.1 implies that the *next* function partitions the set of all binary words of length m into $2^m/2m$ cycles of length $2m$ each. Each such cycle consists of $2m$ words $w_0, w_1, \cdots w_{2m-1}$, where $next(w_i) = w_{i+1(\text{mod } 2m)}$. Furthermore, $w_{i+1(\text{mod } 2m)}$ is obtained from w_i by flipping one bit, and in any m successive applications of *next*, each bit is flipped exactly once.

The following lemma implies that when the value of the counter is given by the function *number*, the *next* function can be used for implementing the **increment** operation.

Lemma 4.2 *Let w be a binary word of length m. Then, $number(next(w)) = number(w) + 1 \pmod m$.*

4.1.2 The Protocol

We are now ready to give a description of the *Cyclic-flip* protocol. The protocol uses m registers. The value of the counter is the non-negative integer we get when applying the *number* function to the content of the m shared registers $r_{m-1}, ..., r_0$. The code for the Cyclic flip protocol is omitted from this abstract. Below we give a description of its operations.

The **look** operation calls the *double-scan* function which reads all the m counter registers sequentially in a cyclic order, until it reads for two consecutive times the same values for all the registers, and

output this m bit word. Then the *number* function is applied to the m bit output of the *double-scan* function and the result is a number v. The output of the `look` operation is the number v.

The `increment` operation uses first the *double-scan* function to get a correct snapshot of the counter register. Then, the *next* function is used to find what bit has to be flipped in order to increment the counter. Finally if nobody else flipped this bit yet then this bit is flipped, otherwise we start all over again.

We point out that the space complexity of the Cyclic-flip protocol (almost) matches the $m-1$ lower bound which follows from the second theorem in the next section.

Notice that in the Cyclic-flip counter, whenever a process completes an `increment` operation, it "knows" the current value of the counter. This fact is used in the solution in the next subsection.

4.2 Dynamic n-counter Protocol

In this subsection we combine the Gray code protocol and the Cyclic-flip protocol to obtain an efficient (unbounded) dynamic n-counter protocol.

Theorem 4.2 *For any n, there is a dynamic n-counter protocol that uses $\alpha(\log m + 1)$ shared registers, where α is the smallest power of 2 that is not smaller than n.*

Let α be the smallest power of 2 that is not smaller than n. Each process starts the execution by assigning itself a unique identifier. This is done using the Cyclic-flip counter protocol described in the previous subsection. We use α registers to implement such a counter. Each process as it wakes up increments this counter by 1, and assigns itself the new value.

Apart from this counter there are n other counters numbered 0 through $n-1$, each consisting of $\log m$ registers. Once a process assigns itself an identifier, say i, it considers counter i as its own local counter that only it can increment and everybody else can read. A process executes an `increment` operation by incrementing its local counter using the Gray code protocol. A process executes a `look` operation using the 4-way scan operation described in the Gray code protocol, on each of the n local counters, and then sums up the results.

5 Lower Bounds

In this section we prove lower bounds on the number of registers needed to implement dynamic counters. Since, by definition, any serial counter is also a dynamic counter, obviously all the lower bounds for dynamic counters hold also for serial counters. Since, when $m = 2$ we can easily solve the counter problem using one shared bit, we assume from now on that $m > 2$. All the results that we prove in this section hold even if it is assumed that `look` operations are atomic. Recall that an *ℓ-bounded* dynamic n-counter is a dynamic n-counter where a total of at most ℓ `increment` operations are performed in any of its runs. Note that $\min\{\log m, \log \ell\}$ is a trivial lower bound on the number of registers needed to implement such a counter. The main result of this section is the following:

Theorem 5.1 *Let $k = \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$. Any ℓ-bounded dynamic n-counter protocol must use at least k registers. This bound holds even when the processes are distinct and there is only one possible initial state.*

Proof: Assume to the contrary that there exists an ℓ-bounded dynamic n-counter protocol that uses $k < \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$ registers. This implies that $n \geq 2k$, $m/2 \geq k$, and $\ell \geq 3k^2$. We show how this assumption leads to a contradiction.

To make the result stronger we assume that there is only one possible initial state. To simplify the presentation we assume that in this sinlge initial state the value of the counter is zero.

For $b \in \{0,1\}$, we say that process p is *b-loaded* for register r in a run x if the event $rmw_p(r, b, \neg b)$ is enabled at x. That is, if p takes a step next, it is going to flip the bit in r from b to $\neg b$. We notice that by axiom *RMW1*, if p is b-loaded for r at x then p is also b-load for r at any run that is indistinguishable to p from x, provided that the value of r at that run is b. We now construct a run ρ as follows:

> We repeat the following procedure at most $2k$ times. Starting with $i = 1$ we let process p_i repeatedly increment the counter until one of the following two situations happens: (1) p_i becomes the first b-loaded process for some register r, (and in

this case we say that p is *suspended* on r); or (2) p_i completes an `increment` operation and the total number of `increment` operations that have begun so far is ℓ. If (1) happens, and we still have not suspended $2k$ processes, then we activate process p_{i+1} according to the above procedure (and increment i by one). The construction terminates when either (2) happens or $2k$ processes have been suspended already.

We note that for each register at most two processes may become *suspended* on it (one may become 0-loaded and the other may become 1-loaded), and since $n \geq 2k$ we have always a process to activate when needed.

We call an `increment` operation that was completed in the run ρ *reversible* if for any register that was changed during this operation there were already two processes suspended on it, and the `increment` operation is *irreversible* otherwise. It is not difficult to see that at most k `increment` operations in ρ are *irreversible*. This follows from the fact that every register can be changed at most once during the run ρ while there is only one process suspended on it.

Next we show that if $2k$ processes are suspended at ρ then we are done. If this is the case, then for any of the k registers there is a 0-loaded process and a 1-loaded process suspended on it. This means that by activating some of these processes we have full control over the value of the counter.

For the rest of the proof, $\oplus$ denotes addition modulo m. Let ρ be such a run, and let $I = \mathit{end}(\rho) \oplus k$, where $\mathit{end}(\rho)$ is the number of increments that have been completed in ρ. W.l.o.g. assume that when the value of the counter is $I \oplus (k+1)$, the values of the counter registers $r_k, ..., r_1$ are $v_k, ..., v_1$, respectively; and that in the run ρ, for each register r_i process p_i is $(\neg v_i)$-loaded for r_i in ρ. Activate all suspended processes except processes $p_1, ..., p_k$ (in some order) and let them terminate. Now we still have the k processes suspended and we can activate all of or subset of them (depending on the current values of the counter registers) and get the counter equal $I \oplus (k+1)$. That is, we can extend ρ to a run ρ' where $\mathit{count}(\rho') = I \oplus (k+1)$ and $\mathit{end}(\rho') = I$. Since at most k processes are suspended in ρ', by *D1*, $\mathit{count}(\rho')$ must lie in the cyclic interval $[I, I \oplus k]$. Since it is assumed that $k \leq m/2$, $I \oplus (k+1)$ does not lie in the cyclic interval $[I, I \oplus k]$, a contradiction.

So, let us assume from now on that at most $2k-1$ processes are suspended at ρ. This implies that the total number of `increment` operations that have begun in ρ is exactly ℓ. Hence, the total number of (completed) reversible `increment` operations is at least $\ell - (2k-1) - k$. We call each maximal interval of ρ which contain only reversible `increment` operation, a *segment*. (I.e., a segment does not contain any irreversible `increment` operations and any operations by suspended processes.) We notice that two consecutive segments are separated by one or more irreversible `increment` operations or operations by (eventually) suspended processes. Since there are at most k irreversible `increment` operations, and at most $2k-1$ processes are suspended at ρ, there are at most $3k$ segments. Since there are at least $\ell - 3k + 1$ reversible `increment` operations, $\ell \geq 3k^2$ and k is an integer, there must be a at least one segment which includes (at least),

$$\left\lceil \frac{\ell - 3k + 1}{3k} \right\rceil \geq \left\lceil \frac{3k^2 - 3k + 1}{3k} \right\rceil = \left\lceil k - 1 + \frac{1}{3k} \right\rceil = k$$

reversible `increment` operations.

Let x and y be two prefixes of ρ $(x \leq y)$ such that $(y - x)$ includes exactly k reversible `increment` operations, does not include any irreversible `increment` operations, all `increment` operations in x and y, except those of processes that are suspended in x, has been completed, and no new processes are suspended in $(y - x)$. Such two runs must exist by the argument in the previous paragraph.

We note that for any register r that is changed during $(y-x)$ it must be the case that two processes are suspended on r at x. This follows from the fact that $(y - x)$ consists of *only* completed reversible `increment` operations.

By definition of reversible `increment` operations we can now activate some of the processes that are suspended at y to get and extension z of y such that the values of all the counter registers are the same in x and z. This implies that $\mathit{count}(x) = \mathit{count}(z)$. As already mentioned, any of the k reversible `increment` operations in $(y-x)$, may only be involved in changing the value of a register for which there is exactly two processes suspended on

it. Thus, it is not difficult to observe that, to get z from y it is enough to activate at most $k-1$ of the $2k-1$ processes that may be suspended at y. Also, the runs x and z are indistinguishable to every process suspended in x that is not involved in some event in $(z-y)$. We notice that any process that is suspended in x is also suspended in y (but not necessarily visa versa). Finally, we extend x and z to runs x' and z' respectively, in exactly the same way, by letting all the suspended process in x that are not involved in some event in $(z-y)$ complete their `increment` operations.

Next we show that at most $k-1$ processes are in the middle of an `increment` operation in z' (and also in x'). From the construction, it must be the case that each of the processes which are still involved in an `increment` operation in z' is suspended in y. Hence it is enough to show that for each register r, there is at most one process that was suspended on r in y and is involved in an `increment` operation in z'. There are two possible cases:

1. There are two processes suspended on r in x. Then at least one of them is not activated in $(z-y)$, and can be activated in $(z'-z)$.

2. Less than two processes are suspended on r in x. As noted above this implies that r was not changed in $(y-x)$ and hence no process that is suspended on r is activated in $(z-y)$. Then, if a process is suspended on r at y it can be activated in $(z'-z)$.

Since at most $2k-1$ processes are suspended at x, at most $k-1$ processes are activated in $(y-x)$. Thus, at most $k-1$ processes are in the middle of an `increment` operation in z' (or in x').

From the construction, using the *RMW* axioms it follows that the values of all the counter registers are the same in x' and z' and thus $count(x') = count(z')$. On the other hand, let $end(x')$ be the number of increments that have been completed in x'. Since at most $k-1$ processes are in the middle of an `increment` operation in x', $count(x')$ must lie in the cyclic interval $[end(x'), end(x') \oplus (k-1)]$. Also, since $end(z') \geq end(x') + k$ and since at most $k-1$ processes are in the middle of an `increment` operation in z', $count(z')$ must lie in the cyclic interval $[end(x') \oplus k, end(x') \oplus (k+k-1)]$. Since, $k \leq m/2$, it must be that $count(z') \neq count(x')$, a contradiction. ∎

By slightly modifying the proof of Theorem 5.1, we can prove the following: Let $k = \min(\frac{m+1}{3}, \frac{n+1}{2}, \sqrt{\ell + 4.25} - 1.5)$. Any ℓ-bounded dynamic n-counter protocol must use at least k registers. (Again, this bound holds even when the processes are distinct and there is only one possible initial state.)

It follows from Theorem 5.1 that any dynamic n-counter protocol must use at least $\min(\frac{m+1}{2}, \frac{n+1}{2})$ bits. Next, by making various restrictions on the way processes may increment the counter, we tighten the lower bound. Recall that a protocol is a *1-flip* protocol if a process may change the value of only one register during a single `increment` operation.

Theorem 5.2 *Let $k = \min(l, n)$. Any 1-flip ℓ-bounded dynamic n-counter protocol must use at least $k-1$ registers.*

Proof: Assume to the contrary that $k-2$ registers are sufficient. Consider a run ρ where $k-1$ different processes increment the counter in a sequential manner, one time each, starting from some arbitrary initial state. Since there are only $k-2$ registers the value of at least one register, say r, is changed at least two times. Let p_1 be the process that was the first to change r and let p_2 be the process that was the second to change r. Let ρ_2 be a prefix of ρ where p_2 just completed its `increment` operation, and let ρ_1 be a prefix of ρ_2 where p_2 is just about to start its `increment` operation.

Next we construct the run ρ_3 as follows. Let p_3 be a process that did not participate in ρ_2. We construct ρ_3 by first activating the processes exactly as in ρ_2 until the point where p_1 is about to change r for the first time. Then we activate p_3 until it is also about to change r. This will happen since processes p_1 and p_3 are identical. We then suspend p_3 and let the run continue as in ρ_2. Finally, after p_2 change the value of r for the second time (and completed its `increment` operation) we activate p_3 and let it changes r for the third time and complete it `increment` operation. Notice that between the point where p_3 was suspended and the point when it was activated again the value of r is changed exactly two times and hence p_3 will not

notice that r has been changed and will change r when it is activated again.

Since in ρ_1 the register r was changed once and in ρ_3 it was changed three times the values of all registers in these two runs are the same and hence $count(\rho_1) = count(\rho_3)$. However, the number of `increment` operation in ρ_3 is greater by exactly two than in ρ_1, and since it is assumed that $m > 2$, we reach a contradiction. ∎

It follows from Theorem 5.2 that there does not exist 1-flip dynamic ∞-counter protocol when using only binary registers. Theorem 5.2 can easily be generalized to show that when $m > v$, any 1-flip ℓ-bounded dynamic n-counter protocol must use at least $(k-1)/(v-1)$ v-valued registers, where $k = \min(l, n)$.

The implementations of the `increment` operation in all the protocols we present in this paper, except the protocol in Section 4.2, have the property of being independent of the specific state of the process that executes them. In every single `increment` operation the final content of the counter is determined only by its initial content. We call such a counter protocol a *memory-less* counter protocol. For such protocols, we have a stronger version of Theorem 5.2.

Theorem 5.3 *Any 1-flip memory-less ℓ-bounded dynamic 2-counter protocol must use at least $\ell - 1$ registers.*

The proof of Theorem 5.3 is similar to that of Theorem 5.2 except that in the construction of the run ρ in the previous proof, instead of activating $k-1$ different processes, we activate only one process and let it increment the counter $k - 1$ times.

We point out that Theorem 5.3 does not contradict Theorem 4.2, since the protocol described in Section 4.2 is not a memory-less or a 1-flip protocol. As in Theorem 5.2, it follows from Theorem 5.3 that there does not exist a 1-flip memory-less dynamic 2-counter protocol when using only binary registers. Finally, we notice that the last two theorems hold also for static counters (the proofs are exactly the same). This fact does not contradict Theorem 2.1, since the Positional protocol is not a *1-flip* protocol.

6 Discussion

We study a new basic problem – the concurrent counter problem – in a model where no assumption is made about the initial state of the shared memory. We design efficient protocols for solving the problem and prove several space lower bounds.

Let the time complexity be the total number accesses to the shared memory in order to complete a (`look` or `increment`) operation. The time complexity of both the `look` and `increment` operations in the Positional protocol is $\log m$. In the Gray-code protocol the time complexity of the `look` operation is $4 \log m$, and of the `increment` operation is 1 apart from the first `increment` which takes $\log m + 1$. As for the Cyclic-flip protocol, in the absence of contention, the time complexity of the `look` operation is m, and of the `increment` operation is $m + 1$. When there is contention the complexity can be in the worst case $m^2 + 2m$ for the `look` operation and $2m^2 + m$ for the `increment` operation. As for the protocol discussed in subsection 5.2, the complexity of the `look` operation is $n \log m$, while that of the the `increment` operation is 1, apart from the first `increment` operation which may take $2m^2 + m + 1$.

There are still many interesting open questions related to concurrent counters. Some of these problems are listed below.

The lower bound in Theorem 5.1 is not tight for ℓ. Improving this lower bound (or the corresponding upper bound) may also help in improving the related time bound, and may have implications on the bounds in [FMRT90].

Generalize the results to counters which use registers of constant size larger than one bit. For instance, implement Gray-codes of alphabets of size grater than 2 for optimal dynamic 1-counters. This seems plausible for alphabets of even size.

Generalize or modify counters to objects that support a wider variety of operations. For example, a natural generalization whose implementation raises non-trivial problems is obtained by extending the counter definition to allow a *decrement* operation, which decreases the value of the counter by one. Another operation is to *reset* the counter to some default value.

Acknowledgement: We would like to thank Hagit Brit for helpful comments.

References

[AG91] J. H. Anderson and B. Grošelj. Beyond atomic registers: Bounded wait-free implementations of nontrivial objects. In *Fifth International Workshop on Distributed Algorithms*, October 1991. Submitted for publication.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, September 1990.

[AH90] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the second Annual ACM symposium on parallel architectures and algorithms*, pages 340–349, July 1990.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, May 1991.

[Dij74] E. W. Dijkstra. Self-stablizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Foundations of Computation Theory*, pages 127–140. Lecture Notes in Computer Science, vol. 158, Springer-Verlag, 1983.

[FMT91] M.J. Fischer, S. Moran, and G. Taubenfeld. Space-efficient asynchronous consensus without shared memory initialization. Technion Report #707, Computer Science Department, The Technion, August 1990. Submitted for publication, 1991.

[FMRT90] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. In *Proc. 22st ACM Symp. on Theory of Computing*, pages 106–116, May 1990. See also Technion Report #644, Computer Science Department, The Technion, August 1990.

[Gar72] M. Gardner. Mathematical games: The curious properties of the Gray code and how it can be used to solve puzzles. *Scientific American*, pages 106–109, August 1972.

[Gil58] E. N. Gilbert. Gray codes and paths on the n-cube. *The bell system technical journal*, 37(9):815–826, May 1958.

[Gra53] F. Gray. Pulse code communication. U. S. Patent 2 632 058, March 1953.

[Koh70] Z. Kohavi. *Switching and Finite Automata Theory.* McGrow-Hill Publication, 1970.

[KMZ84] E. Korach, S. Moran and S. Zaks, Tight lower and upper bounds for some distributed algorithms for complete network of processors. In *Proc. 3th ACM Symp. on Principle of distributed Computing*, pages 199–207, August 1984.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20:806–811, 1977.

[Lam90] L. Lamport. Concurrent reading and writing of clocks. *ACM Trans. on Computer Systems*, 8(4):305–310, 1990.

[Pet82] G. L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. on Programming Languages and Systems*, 4(4):758–762, 1982.

Optimal Multi-Writer Multi-Reader Atomic Register

Amos Israeli
Dept. of Electrical Engineering
Technion — Israel

Amnon Shaham
Dept. of Computer Science
Technion — Israel

Abstract

Two implementations of a multi-writer, multi-reader, atomic register are presented. The physical registers used by the first implementation are single-writer, multi-reader, atomic registers; the physical registers used by the second implementation are single-reader, single-writer, atomic registers. Both implementations are optimal with respect to the two most important complexity criteria: In both implementation the space complexity is *logarithmic*, thus matching the lower bound proven by Cori and Sopena; and the time complexity is *linear*, thus matching the obvious lower bound. These implementations improve upon the space complexity of all previous implementations in their respective classes, by an exponential factor.

1 Introduction

At the most basic level of interprocessor communication, data is transferred via *registers* — memory devices which support *read* and *write* operations. Each register can store any element from its set of *permitted values*; it has a set of *writers* — processors that can write values in the register, and a set of *readers* — processors that can read values from the register. A write operation takes some permitted value as a parameter and stores it in the register; a read operation returns a (permitted) value stored in the register. The stored and returned values should satisfy some consistency guarantees which depend on the type of the register. Lamport in [La86a, La86b] was the first to formalize the notion of a register. He defined three types of registers in terms of the consistency guarantees they supply: *safe*, *regular* and *atomic*. An *atomic register* supports *read* and *write* as atomic, indivisible, operations. Safe and regular registers make inferior consistency guarantees which are not discussed in this paper. The execution of an operation is called an *action*. Under the global time model, which we assume throughout this paper, each action has a starting time and an ending time. The interval between the starting and ending times of an action a is called the *execution interval* of a. Each writer or reader executes actions in a serial manner, but we assume absolutely no synchronization among different processors. Actions of distinct processors might be executed in overlapping time-periods.

In a situation in which some specific type of register is not available at the local hardware store, one may resort to *implementing* the required register using some available registers. For the user, such an implementation is a black box which behaves exactly according to its specifications. Formally, an *implementation* of a *logical* register using a set of *physical* registers, consists of a hardware arrangement of the physical registers and two programs that are called a *writer*

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0071...$1.50

protocol and a *reader* protocol. Both programs are composed of operations of the physical registers which are called *physical* operations and constitute the operations of the logical register which are called the *logical* operations. The set of processors is partitioned into logical writers and logical readers, that is writers and readers of the logical register. For simplicity we assume that each processor is either a logical writer or a logical reader though in reality a processor can function as both. Assuming that each physical register supplies its consistency guarantees, the protocols should satisfy the consistency guarantees of the logical register. In case the logical register is *atomic* it is possible to assign a serialization time for each logical action. The serialization time induces a serialization of all logical actions in the execution. The atomicity of the implemented register implies that every read action returns the value written by the write action which is the most recent preceding write under the induced serialization. To avoid trivial solutions it is required that the reader and writer protocols are *wait-free* and that each action is serialized within its execution interval.

Throughout this paper w and r are used for the number of writers and readers, respectively, and $n = w + r$ is the total number of processors in the system. A register with w writers and r readers is denoted as a (w, r)-register. In [La86a, La86b] Lamport presented five implementations of various logical registers with a single writer. When some of these implementations are combined, they form an implementation of an atomic $(1, 1)$-register with any value set, from binary, safe, $(1, 1)$-registers. Several papers which were motivated by the work of Lamport, studied the intriguing problem of implementing atomic, multi-writer, multi-reader registers. The simplest such an implementation was presented by Vitanyi and Awerbuch in [VA86]. They implement an atomic, (w, r)-register, using atomic, $(1, 1)$-physical registers. In this implementation the physical registers are divided into two fields: a *value* field which stores the value and is not used by the writer and reader protocols, and a *coordination* field which stores all the information needed for the implementation. In previous papers the coordination field is called the *label* field; we reserve the term *label* for the most basic coordination unit and present implementations in which each coordination field consists of several labels. We call implementations in which each register is divided into a value field and a coordination field *label based* implementations. The complexity of a label-based implementation is measured by several criteria. The two most important criteria are:

1. **Space Complexity** - The maximal size of a coordination field of any physical register. (This criterion is often called *label-size*).

2. **Time Complexity** - The maximal number of physical actions executed during a single logical read or write operation.

In the [VA86] implementation labels are timestamps. This causes its main drawback, namely: *unboundedness*. The actual size of a label in any logical action is logarithmic in the number of write actions performed prior to that action. The time complexity of this implementation is linear in n, the total number of processors.

Several researchers have devised bounded label-based implementations for atomic, multi-writer, multi-reader registers, using single-writer, multi-reader physical registers: The first implementation was proposed in [VA86] and was found to be erroneous. The second implementation was proposed by Peterson and Burns in [PB87] — this implementation has a bug which was discovered and corrected by Schaffer in [Sc89]. In this implementation the space complexity is $O(w)$ and the time complexity is $O(w^2)$. Israeli and Li, in [IL87], suggested *bounded time-stamps* as a bounded primitive to capture the temporal relationship among asynchronous processors. Using this method they devised an implementation which runs in linear time and with $O(n)$ space complexity. Another implementation with an inferior complexity is proposed by Abraham in [AB91]. The work of Li, Tromp and Vitanyi in [LTV90] presents an implementation using atomic $(1, 1)$ physical registers. The space complexity is $O(n)$ and the time complexity is

linear. Since the implementation of [LTV90] uses inferior physical registers its complexity is superior to all aforementioned implementations.

Thus far all proposed bounded concurrent implementations have a space complexity which is *linear* in the number of writers, and in some cases even in the total number of processors. Some explanation for this phenomena was suggested by Israeli and Li in [IL87] who defined the class of *Binary Comparison Protocols* (or in short *BCP*) as the class of label-based implementations in which the labels of every two processors can be compared to find the most recent label among the two. They showed that the space complexity of any *BCP* implementation is at least linear in the number of writers. Later Li and Vitanyi in [LV90] have pointed out that though the original [VA86] implementation is *BCP* as well as all bounded implementations, in principal an implementation of an atomic register does not *have to* be *BCP*. To demonstrate this, they presented a *sequential* implementation (in sequential implementations the *logical* actions are executed sequentially, without overlapping) with $O(\log w)$ space complexity. Later it was proven by Cori and Sopena in [CS90] that any implementation for w writers should have at least $2w-1$ distinct labels. They also devised a sequential register with exactly $2w-1$ labels which improved the space complexity of the sequential [LV90] implementation, by a constant.

These works leave an *exponential* gap between the lower and upper bounds on the space complexity of atomic register implementation. While the lower bound relates only to the combinatorial properties of keeping track of the last value (and therefore holds for sequential implementations as well), an actual *concurrent* implementation should also deal with concurrency problems; all existing implementations required *linear* space to do that. The significance of this gap is further emphasized when one takes a closer look at the *unbounded* implementation of Vitanyi and Awerbuch in [VA86]. For polynomial length executions the space complexity of this implementation is *logarithmic*. A linear space complexity is reached by this implementation only in executions of exponential length. In other words: The bounded protocols supersede the unbounded protocol only in *exponentially* long executions. In polynomial length executions, which are often viewed as a better model for real-life situations, an *exponential* overhead is paid for the theoretical boundedness.

The natural question arising here is: *Is this payment necessary?* The answer is *negative* as we show by presenting two bounded, concurrent, label-based implementations for atomic, multi-writer, multi-reader register, with *logarithmic* space complexity. The first implementation uses atomic, $(1,n)$-physical registers. The second implementation uses atomic, $(1,1)$-physical registers. These implementations are the first to break the linear space barrier, for atomic multi-writer registers; moreover, by the lower bound proven by Cori and Sopena in [CS90] the logarithmic space complexity is optimal. The time complexity of both implementations is linear, which is obviously optimal. Both implementations are *self-stabilizing*: Regardless of the system's *initial* state, it eventually reaches a *legitimate* state — a state which can be reached in a legally initialized system. In such an arbitrary initial state the processors may be in arbitrary states (e.g. in the *middle of some logical action*) and the physical registers may hold arbitrary values.

To represent temporal relations among protocol executions we use precedence graphs. These graphs which were introduced by Israeli and Li in [IL87], are time-dependent graphs whose nodes and edges at any given time are determined by the labels stored in the processors' registers at that time. Label ℓ_1 precedes ℓ_2 if there is a directed edge (ℓ_2, ℓ_1) in the precedence graph. Following [ILV87] we use *dynamic precedence trees* in which the outdegree of every node is at most 1, each label points to at most one other preceding label. At any given time the precedence graph is a forest of *intrees* — trees whose edges are directed towards the root. For each individual label *lb* the set of labels whose temporal relationship with *lb* can be found by direct comparison includes the labels whose edges point to *lb*, and the single label to which *lb*'s edge points. Hence our protocols are indeed not *BCP*. Each path of a precedence intree is ordered temporally but labels on distinct

paths are in general not comparable. The paths of any precedence intree are ordered lexicographically, by the *id*s of (the processors which generated) their nodes. The most preceded path in the precedence forest is called the *frontal branch* of the forest. The frontal branch consists of nodes which are generated by recent write actions and the last node on this branch is the last serialized write action.

The rest of this paper is organized as follows: In Section 2 we explain the data structure used by our protocols and present a sequential implementation which serves as an exposition for the ideas which are later used in the concurrent implementations. The $(1, n)$ implementation is presented in Section 3. The $(1, 1)$ implementation is briefly sketched in Section 4, Concluding remarks are brought in Section 5.

2 The Precedence Trees

The three implementations presented in this paper are label-based. Temporal relations among logical actions and among the labels corresponding to these logical actions are represented by use of *precedence graphs* which were originally proposed by [IL87]. These are time dependent directed graphs whose nodes and edges are encoded by the labels generated during the processors' actions. The semantics of the precedence graph is: If there is an edge from label ℓ_1 to ℓ_2 then the action that generates ℓ_1 is serialized after the action that generates ℓ_2. In this paper we use *dynamic precedence trees* which are similar to those used in [ILV87]. In this section we describe the structure of the precedence trees used by all the implementations, and outline a simple sequential implementation of a multi-writer, multi-reader, register using single-writer, multi-reader registers. The sequential implementation does not improve upon the complexity of previously known sequential implementations. It is brought here as an exposition for the ideas which enable the concurrent implementations.

2.1 Data Structure

The writers of the implemented logical registers are denoted by $\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_w$. Execution number a of the writer protocol by $\mathcal{W}_i$ is denoted by L_i^a. Each individual execution of the writer protocol, L_i^a, *generates* a single label which is denoted by ℓ_i^a; i and a are called the *id* and the index of L_i^a (and of ℓ_i^a), respectively. Label ℓ_i^a is stored in the register of $\mathcal{W}_i$, at the end of L_i^a.

All our protocols share an identical structure of the labels which is described below: Each label encodes a node and a potential edge emanating from that node, where the outgoing edge is directed either towards the label itself to form a self loop, or towards nodes encoded by other labels. For convenience we identify the node of ℓ_i^a with the label itself and denote it by ℓ_i^a as well. The node of ℓ_i^a is specified by the number i, which is called the *id* of the node, and by its *address* which is an integer. Since the *id* of all nodes of $\mathcal{W}_i$ is i, the *id* of a node is omitted from its encoding in ℓ_i^a, hence the node of ℓ_i^a is encoded by the *address* field which is denoted by $\ell_i^a.address$. The edge of ℓ_i^a is encoded by the *edge* field which is denoted by $\ell_i^a.edge$. The edge field stores the *id* and *address* of the node to which the potential edge is directed, in two subfields, which are denoted by $\ell_i^a.edge.id$ and $\ell_i^a.edge.address$, respectively. The potential edge emanating from ℓ_i^a *exists* in some precedence graph G if for some label ℓ_j^b in G, it holds that $\ell_i^a.edge = (j, \ell_j^b.address)$. In case this equality does not hold for any node (label) in G, there is no edge outgoing from ℓ_i^a in G. Our protocols make sure that in any precedence graph the aforementioned equality holds for at most one label, hence each label ℓ_i^a encodes at most one outgoing edge which is denoted by $e_i^a = (\ell_i^a, \ell_j^b)$, where ℓ_j^b is the node towards which e_i^a is directed.

We require that $\ell_i^a.edge.id \leq i$, thus the only type of cycles that we permit are self loops. Consequently the nodes (labels) on each directed path of the precedence graph are ordered in increasing order of their *id*s from the root to the leaves. The writers' *id*'s lie between 1 and w. In case $\ell_i^a.edge.id = 0$ we say that e_i^a exists and is directed to the *virtual root label* ℓ_0^0. Since the out-degree of each node is at most 1, each precedence

graph is a forest of labeled intrees — trees whose edges are directed towards the root. From now on we assume that the node set of each precedence graph includes the root label ℓ_0^0. An edge (ℓ_j^b, ℓ_i^a) reflects the fact that L_j^b is serialized after L_i^a. If two edges (ℓ_j^b, ℓ_i^a) and (ℓ_k^c, ℓ_i^a) enter the same node ℓ_i^a, then the serialization of actions L_j^b and L_k^c cannot be determined by (the precedence relation induced by) the edges of the precedence tree. In this case we adopt the common convention and serialize these actions by the *id* of their labels where higher *ids* are serialized *before* lower *ids*. Later we define and use the *history graph* of an execution whose nodes are all labels generated during the execution. In this graph it is necessary to serialize labels with equal *ids* whose edges enter the same node. Since the actions of each individual processor are temporally ordered by their indices, a label with a lower index is serialized before a label with a higher index. These requirement are formally accommodated by the following lexicographic ordering of labels: Let ℓ_i^a and ℓ_j^b be two labels; ℓ_i^a *locally precedes* ℓ_j^b if either $j < i$ or if $i = j$ and $a < b$. Though the *locally precedes* relation is defined for any pair of distinct labels, it reflects a precedence relation *only* among labels whose edges enter the same node in the precedence graph.

Let G be some precedence graph. The *frontal branch* of G is a path which is defined as follows: The first node in the frontal branch is the virtual root label ℓ_0^0; the second node of the frontal branch of G is a node whose edge enters the root and which is *locally preceded* by all other nodes in G whose edges enter ℓ_0^0. In general if α is a prefix of the frontal branch of G whose last node is ℓ_i^a then the next node in the frontal branch is the label whose edge enters ℓ_i^a and which is locally preceded by all other labels whose edges enter ℓ_i^a. The last node in the frontal branch of G (whose indegree is 0), is called the *last* node of G. Let $e_1 = (\ell_i^a, \ell_k^c)$ and $e_2 = (\ell_j^b, \ell_k^c)$ be two edges in some precedence graph G, such that e_1 belongs to the frontal branch of G (that is ℓ_j^b locally precedes ℓ_i^a). In this case we say that e_1 *excludes* e_2 from the frontal branch of G. A pictorial description of a precedence tree appears in Figure 1, where the edges of the frontal branch appear as solid arrows and the rest of the edges appear as dotted arrows. The basic idea in all the implementations is: *At any time t the labels stored in the registers encode a precedence graph whose last node belongs to the last write action completed.*

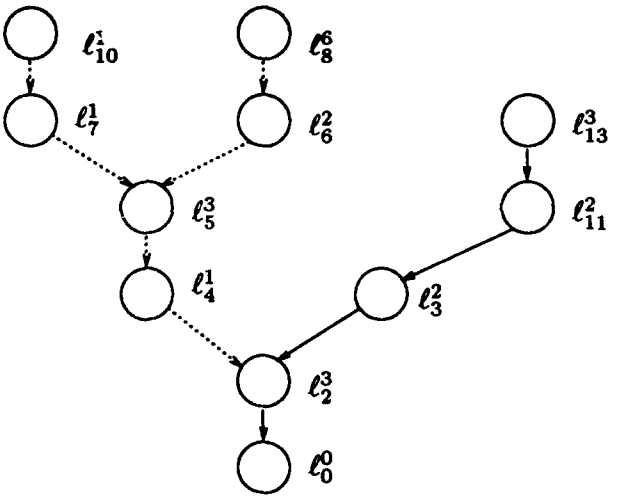


Figure 1: A precedence intree

2.2 Sequential Implementation

Each writer in the sequential implementation writes in a $(1, n)$-register which can be read by all processors, writers and readers. (Since the implementation is sequential we need not require that the registers are atomic.) In this implementation readers do not write. The *current* labels at time t are the labels stored in the processors' registers at time t. The *collect* operation consists of reading the labels of all writers and computing the precedence graph. This precedence graph is called the *current graph*; the precedence tree containing the root label ℓ_0^0 is called the *current tree*. The nodes of the current graph are the root and all the current labels, that is $w + 1$ nodes.

Let α be a path in a precedence tree, the *i-prefix* of α, denoted by α/i, is the prefix of α which contains all nodes whose *id* is less then i. The protocol for $\mathcal{W}_i$, $1 \leq i \leq w$, works as follows: $\mathcal{W}_i$ collects the labels of all writers, computes the current tree and chooses a new label which replaces its previous active label. The edge of the new label is directed towards the last node in the

i-prefix of the frontal branch of the current tree. The new edge excludes the rest of the previous frontal branch from the new frontal branch. At the same time $\mathcal{W}_i$ invalidates all edges which are directed towards its previous label, by choosing a new address which is not the address of the *edge* field of any writer. The reader protocol is simply to collect the current tree and to return its last node. The register of $\mathcal{W}_i$, R_i, is initialized to $((i,0),0)$. At the initial state all edges of the current graph are self-loops and the (initial) current tree contains only the virtual root label ℓ_0^0. The initial logical value is the value corresponding to ℓ_0^0 which can be chosen freely from the permitted values of the logical register.

Correctness of the sequential implementation is straight forward and is left to the reader. In order to allow a writer to invalidate (at most) $w-1$ edges of the precedence tree, it should have at least w possible addresses. Therefore for each writer, the size of the *edge* field is $2\log w$, the size of the address field is $\log w$, and the size of the entire label is $3\log w$.

3 Multi-Writer out of Multi-Reader Registers

In this section we present a concurrent implementation of a (w,r)-atomic register using physical $(1,n)$-atomic registers. In this implementation communication is again one-sided, readers do not write. An important design decision in this implementation is to serialize all logical write actions *independently* of the scheduling of the logical read actions. Serialization of the logical read actions is then done in accordance with the serialization of the write actions. Therefore this section is divided into two subsections: The first subsection presents the writer protocol and the serialization scheme for the logical write actions, and continues with a proof of the correctness of this serialization scheme. The second subsection presents the reader protocol and the serialization scheme for the logical read actions, and then proceeds to prove the correctness of the serialization scheme for the read actions which implies the correctness of the entire implementation.

3.1 The Writer Protocol

3.1.1 Description

The writer protocol is obtained by adjusting the sequential writer protocol to the concurrent environment while keeping its basic ideas and data structure. The structure of the labels in this implementation is identical to the structure of labels in the sequential implementation and once more they encode a labeled intree as a precedence graph whose last node corresponds to the last completed write action. In a concurrent environment however, a writer cannot simply replace its current label by a new label which becomes the last in the current tree. If the protocol were to work in this fashion a new label might be *pointed at* by a label of another write action which is completed at an earlier time, immediately when it is written. In this way the precedence graph may not reflect the temporal order of actions. This problem does not rise in the sequential implementation which supports only non-overlapping logical actions. To overcome this problem the coordination field of a writer's register in this implementation consists of two labels called *new* and *current*. During L_i^a, the *new* field stores a tentative choice for ℓ_i^a, the next label of $\mathcal{W}$ [illegible]inal ℓ_i^a is written in the *current* field in the last physical action of L_i^a. The details of this mechanism are explained below. Accordingly, the structure of R_i, the register of $\mathcal{W}_i$ is: $R_i = (value, new, current)$, and the space complexity is equal to the size of two labels rather than one. Since the value field is not used by the protocol we assume from now on that each label is written with the corresponding value and omit it from the protocol's description.

Similar to the sequential implementation the register of $\mathcal{W}_i$, R_i, is initialized to $((i,0),0,(i,0),0)$. That is, at the initial state all edges of the current graph are self loops, the (initial) current tree contains only the root label ℓ_0^0 and the initial logical value is the value corresponding to ℓ_0^0, which can be chosen freely from the set of all permitted values. Let ℓ_i^a and ℓ_j^b be two labels, the fact that ℓ_i^a and ℓ_j^b are actually the same label, i.e. $i=j$ and $a=b$, is

denoted by $\ell_i^a \equiv \ell_j^b$. Since the relation $\equiv$ is not computable by the processors, the protocol uses the relation $\ell_j^b \simeq \ell_j^c$ which denotes the fact that $\ell_j^b.edge = \ell_j^c.edge$ and $\ell_j^b.address = \ell_j^c.address$.

begin
  Collect G_i
  $last :=$ *id* of the last node of B_i/i
  $new\_address :=$ select (G_i)
  $new\_edge := (last, \ell_{last}.address)$
  $new := (new\_edge, new\_address)$
  $R_i :=$ **write** $(new, current)$
  $n\ell_{last} :=$ **read** $R_{last}.current$
  if $\ell_{last} \simeq n\ell_{last}$
  then
    /* *connect:* Direct e_i^a towards *last* */
    $current := new$
  else
    /* *loop:* Direct e_i^a toward ℓ_i^a */
    $current.edge := (i, new\_address)$
    $current.address := new\_address$
  endif
  $R_i :=$ **write** $(new, current)$
end

Figure 2: The protocol for $\mathcal{W}_i$

The writer protocol appears in Figure 2. We now describe the protocol assuming that L_i^a is executed: First $\mathcal{W}_i$ executes the procedure *collect* in which it reads the registers of all other writers and computes a graph, denoted by G_i^a. The labels of G_i^a are those read from the *current* fields of all registers. Execution of *collect* takes w atomic physical actions which are denoted by $r_i^a[1] \ldots r_i^a[w]$. During collection of G_i^a some writers may change their labels, therefore G_i^a is not necessarily equal to the current graph at any time during the collection of G_i^a. The frontal branch of G_i^a is denoted by B_i^a. The node with the maximal *id* in B_i^a/i is the *last* node of B_i^a/i. Let ℓ_{last} be the last node of B_i^a/i. After G_i^a is collected, $\mathcal{W}_i$ chooses a tentative *new* label whose edge is directed towards ℓ_{last} and whose address is obtained by the function *select*; this function ensures that the *new* label is not the head of any (existing or potential) edge encoded by any *current* label or *new* label read during L_i^a. In this way all edges entering ℓ_i^{a-1} are invalidated. In addition the function *select* ensures that $new\_address \neq \ell_i^{a-1}.address$ even if no other edge is directed towards ℓ_i^{a-1}; thus a writer never uses the same address twice in a row. The chosen label is declared by $\mathcal{W}_i$ by writing it into the *new* field of R_i while the *current* label is not changed. The declaring write action is denoted by d_i^a. Following d_i^a, $\mathcal{W}_i$ rereads the register of $\mathcal{W}_{last}$, the writer whose label is *last* in B_i^a/i. This second read action is denoted by $\tilde{r}_i^a[last]$. The label read in action $\tilde{r}_i^a[last]$ is called the *target* label of L_i^a. The logical write action L_i^a is concluded by a final physical write action which is denoted by w_i^a. In this action $\mathcal{W}_i$ replaces its *current* label as follows: Let ℓ_{last} and $n\ell_{last}$ be the two labels read by $\mathcal{W}_i$ in actions $r_i^a[last]$ and $\tilde{r}_i^a[last]$ respectively. If $\ell_{last} \simeq n\ell_{last}$ then *new* label is assigned to *current*, in this case we say that L_i^a *connects*. If however $\ell_{last} \not\simeq n\ell_{last}$ then *current* is chosen such that its edge is a self-loop directed towards ℓ_i^a itself. In this case we say that L_i^a *loops*.

3.1.2 Serialization Scheme for Logical Write Actions

The atomicity of an implementation is proved by showing that for every physical sequential execution, the induced logical execution can be serialized. A sequential execution is entirely determined by its schedule; the complete asynchrony of the system means that the only points in time which can be used to serialize a logical action, are the occurrence times of the physical actions, and whenever some flexibility is possible, the intervals between these occurrence times. For this reason we sometimes use the name of an action to denote its occurrence time. Whenever we say that some property p holds at action a we mean that p holds right after the occurrence time of a. In the following subsections we fix some arbitrary sequential physical execution with respect to which all definitions and proofs are being made, since this fixed execution is arbitrary, the results hold for every system execution of the

implementation. The logical write actions are serialized by an explicit assignment of *serialization time* for each action. This assignment is done using a *History graph* — a precedence graph whose structure reflects the execution of the system. The history graph — which is not computable by the processors — plays a key-role in the correctness proof of our protocols.

Definition 1: Let E be an execution of the system. The *History graph* of E, H, is a time dependent graph which is constructed incrementally during the execution, as follows:

1. H^0 is the *History graph* at time 0 (before the execution begins). It contains only the *root* node — ℓ_0^0.
2. Let L_i^a be an arbitrary logical write action. Let t be the time when w_i^a occurs, the time before t and after any other physical action which precedes w_i^a is denoted by by t^-. H^t is the graph obtained from H^{t^-} by adding the node, ℓ_i^a, and the directed edge e_i^a as follows: If L_i^a connects then e_i^a is directed towards the target label of L_i^a, otherwise (L_i^a loops) e_i^a is a self loop. According to our convention of using an action's name to denote its occurrence time, H^t is also denoted by $H^{w_i^a}$.

At any time t, the history graph H^t is a precedence forest which consists of a single precedence tree and some disjoint self-loops. The frontal branch of H^t is denoted by B_H^t. In order to serialize the logical actions we first partition the set of logical write actions into two subsets, *good* and *bad*, where a good write action is a write action whose label is last in B_H^t at the time t that it joins the history graph.

Definition 2: Let L_i^a be an arbitrary logical write action Action. L_i^a is *good* if ℓ_i^a is *last* in $B_H^{w_i^a}$. A logical write action which is not good, is *bad*.

Obviously if L_i^a loops then it is bad, but there are many cases in which L_i^a connects and it is also bad. Using the partition of logical write actions to good and bad we assign a *serialization time* to every logical write action. The serialization time establishes a total order on the set of logical write actions:

Definition 3: Let t be the occurrence time of w_i^a. Define the serialization time of L_i^a as follows:

1. If L_i^a is good, that is, ℓ_i^a is the last node of B_H^t, then L_i^a is serialized at t.
2. If L_i^a is bad and ℓ_j^b is the last node of B_H^t then L_i^a is serialized before L_j^b and after any other physical action which precedes w_j^b. In this case we say that L_i^a is *serialized by* L_j^b. In case two bad write actions, L_i^a and L_j^b, are serialized by the same good write action L_k^c, L_i^a and L_j^b are serialized by their *ids*: The action with the lowest *id* first, and the action with higher *id* second.

Under the defined serialization time, at the occurrence time of every physical action, t, the last node in B_H^t is the most recent write action, that is the value of the logical register.

3.1.3 Correctness of the Serialization Scheme for Logical Write Actions

In the correctness proof we have to prove that the serialization time satisfies the serialization requirements for atomic registers. In Theorem 9 we prove that every logical write action is serialized within its execution interval. In Theorem 11 we prove that all graphs collected by the writers are precedence graphs. We start the correctness proof with some technical lemmas.

Lemma 1: Let ℓ_i^a be a node in H^t. If $\ell_i^a \notin B_H^t$, then for any $t' > t$, $\ell_i^a \notin B_H^{t'}$.

Let a and b be two physical atomic actions; the fact that a occurs before b is denoted by $a \rightarrow b$.

Lemma 2: If (ℓ_i^a, ℓ_j^b), $i \neq j$, is an edge in H then w_j^b occurs before w_i^a (i.e. $w_j^b \rightarrow w_i^a$).

Lemma 3: If $\ell_i^a \in B_H^t$ then L_i^a is good.

Lemma 4:

(a) The sequence of node *ids* along any path of the history graph from the leaf towards the root is strictly decreasing.

(b) Every path of the *history graph* contains at most one label of every writer.

Due to concurrency it is not guaranteed that the edges of a graph collected by any writer or reader belong to the history graph. A weaker relationship between the edges of the collected graphs and the edges of the history graph is depicted in the next lemma:

Lemma 5: If (ℓ_i^a, ℓ_j^b) is an edge in G_k^c then there exists an integer r, $r \geq 0$, such that (ℓ_i^a, ℓ_j^{b+r}) is an edge in H.

The next lemma proves that if ℓ_i^a belongs to the frontal branch of the history graph then it is the current label of $\mathcal{W}_i$.

Lemma 6: If $\ell_i^a \in B_H^t$ for some time t, then ℓ_i^a is the current label of $\mathcal{W}_i$ at t.

The next corollary forms a tighter relationship between edges of G whose head belongs to the frontal branch of the history graph and the corresponding edges of H.

Corollary 7: Let (ℓ_i^a, ℓ_j^b) be an edge in G_k^c. If $\ell_j^b \in B_H^{r_i^a[j]}$ then (ℓ_i^a, ℓ_j^b) is an edge in $H^{r_i^a[j]}$.

The fact that every label $\ell_j^b \in B_H^t$ is in B_i^a and every label $\ell_k^c \in B_i^a$ is in B_H^t, is denoted by $B_H^t \equiv B_i^a$. In the next lemma and in the theorem that follows, we prove that the serialization time of each logical write action lies within its execution interval.

Lemma 8: Let t_0 be the occurrence time of $r_i^a[j]$. If at $t \geq t_0$ $B_H^t/(j+1) \not\equiv B_i^a/(j+1)$, then there exists a good logical action L_k^c, for some $k \leq j$, such that w_k^c occurs within the time interval starting at $r_i^a[k]$ and ending at t.

Theorem 9: Every logical write action is serialized within its execution interval.

The fact that L_i^a is serialized before L_j^b is denoted by $L_i^a \Rightarrow L_j^b$. The next lemma proves that the history graph is a precedence graph with respect to the relation $\Rightarrow$. Since the history graph is not computable by the processors, the significance of this lemma is reflected in the theorem that follows in which it is proved that the graphs collected by the processors are also precedence graphs with respect to the relation $\Rightarrow$.

Lemma 10: If (ℓ_i^a, ℓ_j^b), $i > j$, is an edge in H then $L_j^b \Rightarrow L_i^a$.

Theorem 11: If (ℓ_i^a, ℓ_j^b), $i > j$, is an edge in G_k^c then $L_j^b \Rightarrow L_i^a$.

3.2 The Reader Protocol

3.2.1 Description

Like the writer protocol, the reader protocol is obtained by adjusting the sequential reader protocol to the concurrent environment. Though the basic idea in this implementation is to keep a precedence tree whose last node is the last value written to the logical register, it is not possible to just read the current tree and return its last node: Due to concurrency, the current tree and its last node may change during the execution of the collect procedure by the reader. In particular a label of an action which should not be returned by a reader, may appear as last in its collected tree. This happens when some concurrent write actions cause the reader to see some branches of the tree as "hanging in the air". Therefore a single collection is not sufficient. Our protocol collects three forests and analyzes the differences among these forests to determine the returned label. The three forests are denoted by G, $\overleftarrow{G}$, and $\tilde{G}$. Forest $\overleftarrow{G}$ is collected in reverse order — from R_w down to R_1, therefore most of the lemmas proved in the previous section do not hold for $\overleftarrow{G}$. The analysis of the three graphs does not yield

an accurate description of the current graph, but rather enables the reader to identify a label which satisfies the requirements for the reader protocol as follows: The identified label is either last in the history graph when the logical read action starts, and hence it is the last logical write action that is serialized before the read action starts, or the identified label is generated by a logical write action serialized within the execution interval of the logical read action. In this case the logical read is serialized immediately after the logical write. The physical actions, executed by $\mathcal{R}_u$, during the reader protocol are denoted by: $r_u[1] \ldots r_u[w]$, in which G is collected, $\overleftarrow{r}_u[w] \ldots \overleftarrow{r}_u[1]$, in which $\overleftarrow{G}$ is collected, and $\tilde{r}_u[1] \ldots \tilde{r}_u[w]$, in which $\tilde{G}$ is collected. The notation $B_i^a \subset B_j^b$ is used when for each $\ell_k^c \in B_i^a$, there is a label $\ell_k^d \in B_j^b$ such that $\ell_k^c \simeq \ell_k^d$. The notation $B_i^a \simeq B_j^b$ is used when $B_i^a \subset B_j^b$ and $B_j^b \subset B_i^a$. The code of the reader protocol appears in Figure 3.

begin
  Collect G_u; Collect $\overleftarrow{G}_u$; Collect $\tilde{G}_u$
  last := *id* of the last node of $\tilde{B}_u$
  if $(B_u \simeq \tilde{B}_u \not\simeq \overleftarrow{B}_u)$ **then**
    $i := \min_j\{(\ell_j^b(\in G_u) \not\simeq \ell_j^c(\in \overleftarrow{G}_u))$ **and** $(\ell_j^c \not\simeq \ell_j^d(\in \tilde{G}_u))\}$
    return ℓ_i^d
  elseif $(B_u \simeq \overleftarrow{B}_u \simeq \tilde{B}_u)$ **then**
    return the label of *last* in $\tilde{B}_u$
  elseif $(B_u \not\simeq \tilde{B}_u)$ **and** $(B_u \subset \tilde{G}_u)$ **then**
    return the label of *last* in $\tilde{B}_u$
  elseif $(B_u \not\simeq \tilde{B}_u)$ **and** $(B_u \not\subset \tilde{G}_u)$ **then**
    $i := \min_j(\ell_j^a(\in B_u) \not\simeq \ell_j^d(\in \tilde{G}_u))$
    return ℓ_i^d
  endif
end

Figure 3: The protocol for $\mathcal{R}_u$

3.2.2 Serialization Scheme for Logical Read Actions

Throughout this paper S_u^a denotes the a-th execution of the read protocol by $\mathcal{R}_u$. The serialization time of any logical read action is determined by the serialization time of the logical write action whose value is returned by the read action, according to the following proposition:

Proposition 12: If for any logical read action S_u^a the returned label ℓ_j^b satisfies one of the following conditions:

1. L_j^b is the logical write action that is serialized last before S_u^a.
2. L_j^b is serialized within the execution interval of S_u^a.

then the implementation is atomic.

To prove the proposition correct we have to show that if for every action in some execution E, one of these conditions holds, then the E is serializable. This is proven by the following serialization scheme:

Definition 4: Let ℓ_i^b be the label returned by S_u^a. Denote by t_s and t_e the occurrence time of $r_u^a[1]$ and $\tilde{r}_u^a[w]$ respectively. The serialization time of S_u^a is defined as follows:

1. If L_i^b is not serialized within the execution interval of S_u^a then S_u^a is serialized at t_s.
2. If L_i^b is serialized at time t which lies within the execution interval of S_u^a then S_u^a is serialized at t^+ where t^+ denotes the time immediately after t.

3.2.3 Correctness of the Implementation

The correctness of the serialization scheme for logical read actions, and the correctness of the entire implementation, is based on Proposition 12 and on the following theorem:

Theorem 13: Let (t_s, t_e) be the execution interval of S_u, let ℓ be the label returned by S_u, and let L be the logical write that produced the label ℓ. L satisfies one of the following two claims:

1. L is the last logical write action serialized before t_s (and hence ℓ is last in $B_H^{t_s}$), or
2. L is serialized within the interval (t_s, t_e).

4 Multi-Writer out of Single-Reader Registers

In this section we briefly sketch an implementation in which the physical registers are atomic $(1,1)$-registers. Though this implementation follows the lines of the previous implementation, the use of inferior physical registers requires some adjustments. First, every single physical action in the previous implementation is now replaced by n physical actions (for example w_i^a is replaced by $w_i^a[1]$ ...$w_i^a[n]$.) Second, in this implementation communication is two sided: Every pair of processors, P_i and P_j, (regardless whether they are writers or readers) communicate via a pair of atomic $(1,1)$-registers. The writer protocol follows the lines of the protocol in the $(1,n)$ implementations with few minor changes. In the reader protocol we take advantage of the use of $(1,1)$ registers to decide upon the returned value after collecting a single graph. In an implementation based on $(1,1)$ registers, a reader can identify executions of logical actions whose execution interval is *enclosed* within the execution interval of the reader, without enlarging the label-size over $O(\log n)$ bits. This is done by using well known hand-shake mechanism. The main idea behind the reader protocol is as follows: If the reader does not see any enclosed label during the collection of its graph G, then the frontal branch of the history graph before the collection starts is a subgraph of G. Therefore if there exists an enclosed label then the reader returns it; otherwise the reader returns the last label of G.

The serialization time of all write actions is determined once more using the history graph. The nodes of the history graph are labels of writers and (unlike the previous implementation) of readers. The outgoing edge from ℓ_i^a in H, e_i^a, is determined in the same way it is done in the previous implementation. The time ℓ_i^a joins the history graph always falls between $w_i^a[1]$ and $w_i^a[n]$: Let t be the first time that some label joins the history graph with outgoing edge directed towards ℓ_i^a. If $w_i^a[n]$ occurs after t then ℓ_i^a joins H at t^-; otherwise ℓ_i^a joins H at the occurrence time of $w_i^a[n]$. Let L_i^a be a logical write action and let t be the time ℓ_i^a joins the history graph. If L_i^a is good (that is last in B_H^t) it is serialized at t. If L_i^a is bad it is serialized by the label that was last in $B_H^t/(w+1)$.

The serialization time of a logical read action S_u^a is determined by the write action, L_i^b, whose value is returned by S_u^a. It can be shown that L_i^b is either serialized within the execution interval of S_u^a, or that it is the last write action serialized before S_u^a begins. In the first case S_u^a is serialized just after L_i^b. In the second case S_u^a is serialized at the beginning of its execution interval.

5 Concluding Remarks

We have presented two implementations of a multi-reader, multi-writer, atomic register. Both implementations use a novel method of dynamic precedence trees in which only partial precedence information is represented and therefore they are not *BCP*. Both implementations are optimal with respect to the most important complexity criteria: They have logarithmic space complexity and linear time complexity. Communication in the multi-reader registers based implementation is one sided, only writers execute physical write actions. In the single-writer register based implementation communication is two-sided. Recently it was proved by Israeli, Tromp and Vitanyi in [ITV92] that there exists no such implementation with one-sided communication. The existence of an implementation which is based on $(1,1)$ atomic registers with label-size which depends only on the number of writers remains open.

acknowledgements

We thank Paul Vitanyi for his tireless efforts to convince us that implemetations with sublinear space complexity are worth looking at. We also thank Yael Gafni, Arie Rudich and John Tromp whose comments on an earlier vrsion have helped us in this presentation.

References

[AB91] U. Abraham, "MultiWriter Atomic Registers and bounded timestamps," preprint.

[CS90] R. Cori and E. Sopena "Some combinatorial aspects of Time Stamps Systems," submitted to J. Of Algorithms.

[IL87] A. Israeli, and M. Li, "Bounded Time-Stamps," Proceedings of the 28th Annual Symposium on Foundations of Computer Science, 1987, pp. 371-382.

[ILV87] A. Israeli, M. Li, and P. Vitanyi, "Simple Multireader registers using Time-Stamp schemes," Report no. CS-R8758 Center for Mathematics and Computer Science, Amsterdam, Holland, November 1987.

[ITV92] A. Israeli, J. Tromp, and P.M.B. Vitanyi, personal communication.

[La86a] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism," Distributed Computing 1, 2 1986, pp. 77-85.

[La86b] L. Lamport, "On Interprocess Communication. Part II: Algorithms," Distributed Computing 1, 2 1986, pp. 86-101.

[LTV90] M. Li, J. Tromp, and P.M.B. Vitanyi, "How to Share Concurrent Wait-Free Variables," Tech. Rept. CS-R8916, CWI, April 1989. Submitted to J.ACM/revision. (Prelim. Version in ICALP89)

[LV90] M. Li and P.M.B. Vitányi, "Optimality of Wait-Free Atomic Multiwriter Variables," Submitted to Information Processing Letters 1990. Techn. Rept. CS-R9128, CWI, June 1991.

[PB87] G.L. Peterson and J.E. Burns, "Concurrent reading while writing II: the multiwriter case", 28th Annual IEEE Symp. on Foundations of Computer Science, 1987, pp. 383-392.

[Sc89] R.W. Schaffer, "On the Correctness of Atomic Multi-Writer Registers", preprint.

[VA86] P. Vitanyi, and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," Proceedings of the 27th Annual Symposium on Foundations of Computer Science, 1986, pp.233-243.

Tolerating Linear Number of Faults in Networks of Bounded Degree

Eli Upfal *

Abstract

In [7], Dwork et al. proposed a new paradigm for fault tolerant distributed computing termed *almost everywhere agreement.* While all other fault tolerance paradigms require networks of high connectivity to tolerate substantial number of faults, it was shown in [7] that the new paradigm can be achieved even on bounded degree networks, as long as the number of faults is bounded by $O(n/\log n)$, where n is the size of the network.

A major problem that was left open in [7] is whether almost everywhere agreement can be achieved on bounded degree networks in the presence of up to $O(n)$ faulty nodes (processors). In this work we answer this question in the affirmative. As in [7], our solution is based on a general technique for simulating on a bounded degree network an algorithm designed for the complete network. Each communication round of the complete network protocol is simulated by a logarithmic number of communication rounds, and with a polynomial number of messages.

1 Introduction

Achieving processor cooperation in the presence of faults is a major problem in distributed systems. Consider a network in which each node is a processor and each edge is a communication link. Popular paradigms such as Byzantine agreement require $\Omega(t)$ connectivity in the communication network in order to tolerate t faults [5, 9]. A simple corollary of this result is that a system can reach agreement in the presence of t faulty nodes, only if every processor is directly connected to at least $O(t)$ others. Such high connectivity, while feasible in a small system, cannot be implemented at reasonable cost in a large system.

As technology improves, increasingly large distributed systems and parallel computers will be constructed. In any forthcoming technology, the number of faulty processors in a given system will grow with the size of the system, while the degree of the interconnection network will, for all practical purposes, remain fixed.

Despite these negative observations, distributed systems are widely used and parallel computers are being built. This suggests that the correctness conditions for Byzantine

*IBM Almaden Research Center, San Jose, CA 95120, and Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0083...$1.50

agreement are too stringent to reflect practical situations. In particular, Byzantine agreement guarantees coordination among all correct processors, omitting only the t faulty processors. In many situations it may suffice to guarantee agreement among all but $O(t)$ processors. In other situations a simple majority consensus may suffice. Similarly, in clock synchronization, or in firing squad synchronization, it may suffice for a vast majority of the correct processors to be synchronized.

Motivated by the need to run fault tolerant computing on sparse networks, Dwork et al. [7] introduced the notion of *almost everywhere agreement* (denoted *a.e.-agreement*), in which all but a small number of the correct processors must choose a common decision value. Dwork et al. showed that by relaxing the correctness condition from agreement between all the non-faulty processors, to agreement between almost all the non-faulty processors, one can eliminate the costly connectivity requirement. In particular they show that there are bounded degree networks of $n+O(t)$ processors that guarantee agreement among n correct processors in the presence of up to t faults. Further works by Berman and Garay [3, 4] improve the efficiency of the protocols for achieving the distributed agreement. The a.e.-agreement paradigm admits deterministic solutions in networks of small constant degree to such fundamental problems as atomic broadcast, Byzantine agreement, and clock synchronization.

More precisely, a protocol P is said to achieve $f(t)$-agreement if in every execution of P in which at most t processors fail, at least $n-f(t)$ non-faulty processors eventually decide on a common value. Moreover, if all the correct processors share the same initial value, then that must be the value chosen. Note that the traditional Byzantine agreement problem is just t-agreement.

A protocol P achieves *a.e.-agreement* in the presence of t faulty nodes, if it achieves $f(t)$-agreement for some $f(t) \leq \mu t$, where μ is a constant that is independent of t, and of the size of the network.

The main result in [7] is a bounded degree network and a communication protocol for that network that achieves a.e.-agreement in the presence of up to $O(n/\log n)$ faults. The major problem left open in [7] is whether a.e.-agreement can be achieved on bounded degree networks in the presence of more than $n/\log n$ faults. Note that this problem is significantly harder. In bounded degree networks the distance between most pairs of nodes is $\Omega(\log n)$. Thus, in the presence of more than $n/\log n$ faulty nodes, most communication paths between most pairs of processors include at least one faulty node.

In this work we show that the above difficulty can be overcome and that there are bounded degree networks for which a.e.-agreement is achievable in the presence of up to $O(n)$ faulty processors. Our solution is based on special resilient properties of expander graphs. To simplify the presentation no attempt is made here to compute the best constants. Our goal is to demonstrate the rather surprising fact, that a.e.-agreement in the presence of up to a linear number of faults is feasible on some bounded degree networks.

We give an explicit construction of an n node bounded degree network G, and a communication protocol between pairs of nodes in G. We show that for any set T of faulty nodes, $|T| \leq \alpha n$, the communication protocol guarantees reliable communication between all pairs of processors in a set $P(T)$ of at least $n - \mu t$ non-faulty processors in G (α and μ are constants independent of n and t). The communication protocol requires $O(\log n)$ communication rounds and a polynomial number of messages. Given any protocol designed for a complete network, it can be simulated by our communication protocol on the bounded degree network G, achieving agreement between a set of at least $n - \mu t$

non-faulty nodes in the presence of t faulty nodes.

Our model of computation is identical to that commonly used in the Byzantine literature. Specifically, each processor can be thought of as a (possibly infinite) state machine with specific registers for communication with the outside world. The processors communicate by means of point-to-point links, which are assumed to be completely reliable. The entire system is synchronous, and can be thought of as controlled by a common clock. At each pulse of the common clock a processor may send a message on each of its incident communication links (possibly different messages on different links). Messages sent at one clock pulse are delivered before the next pulse. Note that this model counts communication rounds, and ignores the local computation in the nodes. An intriguing open problem is to construct an efficient algorithm for the computation that each node performs in the process of achieving a.e-agreement. (The local computation required by our solution is super-polynomial in n.)

Byzantine agreement on bounded degree networks in the presence of random faults has been studied by M. Ben-Or and D. Goldriech [2, 8]. They presented a network and a polynomial time agreement protocol for that network that achieves a.e-agreements with high probability in the presence of linear number of random faults in the model where processors fail independently with some fixed constant probability.

2 Combinatorial Characterization of Resilient Networks

Dwork et al. [7] derived the following combinatorial characterization of networks that admit $f(t)$-agreement, for any function $f(t)$. We present the characterization in this section, and prove in the next section that there are bounded degree networks that satisfy it for t up to linear in the size of the network, and $f(t)$ linear in t.

For any agreement protocol P, let $P(T)$ be any maximal set of correct processors that always reach agreement under the protocol P, independent of the behavior of the processors in T (thought of as faulty).

Theorem 1 *[7] Let G be a communication graph, let $\{T_i\}_{i=1}^{k}$ be the family of all possible sets of faulty processors in G, and let $\{A(T_i)\}_{i=1}^{k}$ be a family of sets of processors in G. There exists a protocol P s.t. $P(T_i) = A(T_i)$ for $i = 1, \ldots, k$, if and only if for every pair of processors $u, v \in A(T_i) \cap A(T_j)$, the set $T_i \cup T_j$ does not disconnect u from v in G.*

Sketch of the proof: To prove necessity, we show that if there exist sets T_i and T_j which jointly (but not individually) can disconnect correct processors u and v, then there exist two scenarios, indistinguishable to v, such that in one scenario T_i is faulty and u decides on a value a, while in the second scenario T_j is faulty and u decides on a value b.

We prove sufficiency by constructing a reliable communication protocol between pairs of processors in $A(T)$. We briefly describe a few points of our construction.

A processor u transmits a message to v by sending it along all simple paths from u to v. As the message passes from site to site, each processor appends the name of the processor from which the message was received. Thus, a message that passes through faulty processors contains the name of at least one such processor (the last one). Processor v searches for a set T_i such that all the messages not passing through this set are consistent and both u and v are in $A(T_i)$. Let T be the set of faulty processors in a particular execution of our algorithm. If u and v are in $A(T)$

then v will try this set and extract the correct value. Crucial to our algorithm is that v will never extract an incorrect value. This is because by assumption, for all other relevant sets T_j, $T \cup T_j$ will not disconnect u from v. Thus, v receives the message via at least one fault-free path. Therefore, the faulty processors can at most create an inconsistent set of values, from which v extracts nothing. □

3 Proof of the Main Result

Theorem 2 *There exist*

1. *Constants $\alpha > 0$ μ, and d, independent of t and n;*
2. *An n-vertex d-regular network G, which can be explicitly constructed;*
3. *A communication protocol P;*

Such that for any set of faulty nodes T in G, the communication protocol guarantees reliable communication between all pairs of nodes in a set of non-faulty nodes $P(T)$, such that $|P(T)| \geq n - \mu t$. Furthermore, the protocol requires $O(\log n)$ communication rounds, and generates polynomial (in n) number of messages.

Proof: Given an n-vertex d-regular graph $G = (V, E)$, let $A(G)$ denote the n by n adjacency matrix of G. Clearly d is the largest eigenvalue of $A(G)$. Let $\lambda(G)$ denote the maximum absolute value of any other eigenvalue of $A(G)$. Lubotzky et al. [11] gave explicit construction of d-regular graphs with $\lambda(G) \leq 2\sqrt{d-1}$, for any $d = p + 1$, p prime. We prove that a.e.-agreement in the presence of up to αn faults can be achieved on an n-vertex d-regular network G, with $\lambda(G) \leq \sqrt{d-1}$.

For each set T of faulty processors we define the set $P(T)$ as the outcome of the following procedure:

FUNCTION P
Input: a graph $G = (V, E)$, a set $T \subset V$.

1. $Z \leftarrow \emptyset$;
2. $ADD = \{v \mid v \notin T \cup Z,\ v$ has at least d/5 neighbors in $T \cup Z\}$;
3. WHILE $ADD \neq \emptyset$ DO
   (a) $Z \leftarrow Z \cup ADD$;
   (b) $ADD = \{v \mid v \notin T \cup Z,\ v$ has at least d/5 neihbors in $T \cup Z\}$;
4. END WHILE
5. $P(T) \leftarrow V \setminus (Z \cup T)$;
6. END FUNCTION;

We first show that the set $P(T)$ defined by the above function is sufficiently large.

Lemma 1 *There exist constants $\alpha > 0$, μ, and d, and an n-vertex d-regular graph $G = (V, E)$, such that for every set $T \subset V$, $|T| \leq \alpha|V|$, the set $P(T)$ defined by the above function is greater than $n - \mu|T|$.*

Proof: We need to show that at the end of the execution of the function $P(T)$, $|T \cup Z| \leq \mu t$, for some constant μ.

Alon and Chung [1] gave the following relation between the eigenvalues of a graph and the density of its induced subgraphs: Let $e(S)$ denote the number of edges in G connecting vertices in S, then for any subset $S \subset V$, $|S| = \theta n$,

$$|e(S) - d\theta^2 n/2| \leq \lambda(G)\theta(1-\theta)n/2. \quad (1)$$

Fix $\alpha = 1/72$, $\mu = 6$, and pick an n-node d-regular network $G = (V, E)$ with

$$\lambda(G) \leq 2\sqrt{d-1}.$$

Assume that there is a set $T \subset V$, such that $t = |T| \leq \alpha n$, and $|T \cup Z| > \mu t$. Consider the execution of the function $P(T)$. When a vertex is added to Z it adds $d/5$ edges to the subgraph induced by $T \cup Z$. Since we can add the vertices to Z one at a time, if $|T \cup Z| > \mu t$, then the graph G has a subset of size $\ell = \lfloor \mu t \rfloor$ with at least $(\ell - t)d/5$ internal edges. But since $\mu t/n \leq 1/12$,

$$(\ell - t)d/5 \geq dt - d/5 >$$

$$(1/12)6td/2 + \sqrt{d-1}\mu t \geq$$

$$d(\mu t/n)^2 n/2 + (\mu t/n)\sqrt{d-1}(1 - (\mu t/n))n$$

for sufficiently large (constant) d, which violates (1). □

Lemma 1 shows that for each set T, $|T| \leq \alpha n$, the set $P(T)$ has at least $n - \mu|T|$ vertices. We now need to show that the family of sets $\{P(T) \mid T \in V \ \ |T| \leq \alpha n\}$ satisfies the condition of theorem 1. We prove a stronger result:

Lemma 2 *Given any two sets T_1, T_2 in G, such that $T_i \subset V$, and $|T_i| \leq \alpha n$, for $i = 1, 2$. Any two vertices $v_1, v_2 \in P(T_1) \cap P(T_2)$ are connected by a path of length $O(\log n)$ in the subgraph induced by $V \setminus (T_1 \cup T_2)$.*

Proof: Consider the following variant of the FUNCTION P in which vertices are added to Z when they have $2d/5$ neighbors in $T \cup Z$ instead of $d/5$:

FUNCTION P'

Input: a graph $G = (V, E)$, a set $T \subset V$.

1. $Z \leftarrow \emptyset$;
2. $ADD = \{v \mid v \notin T \cup Z,\ v$ has at least 2d/5 neihbors in $T \cup Z\}$;
3. WHILE $ADD \neq \emptyset$ DO
   (a) $Z \leftarrow Z \cup ADD$;
   (b) $ADD = \{v \mid v \notin T \cup Z,\ v$ has at least 2d/5 neihbors in $T \cup Z\}$;
4. END WHILE
5. $P'(T) \leftarrow V \setminus (Z \cup T)$;
6. END FUNCTION;

Let Z_t^i denote the variable Z after the i-th iteration of computing $P(t_i)$, let Z_t' denote the variable Z after the t-th iteration of computing $P'(T_1 \cup T_2)$.

We prove by induction on t that $Z_t' \subseteq Z_t^1 \cup Z_t^2$. The claim clearly holds for $t = 0$. Assume that the claim holds for $t - 1$, and assume that u was added to Z_t' in the t-th iteration, then u has $2d/3$ neighbors in $T_1 \cup T_2 \cup Z_{t-1}^1 \cup Z_{t-1}^2$, and at least $d/3$ neighbors in either $T_1 \cup Z_{t-1}^1$ or in $T_2 \cup Z_{t-1}^2$. Thus u is in $Z_t^1 \cup Z_t^2$.

Since $Z_t' \subseteq Z_t^1 \cup Z_t^2$,

$$P'(T_1 \cup T_2) \supseteq P(T_1) \cap P(T_2).$$

Furthermore,

$$P'(T_1 \cup T_2) \cap (T_1 \cup T_2) = \emptyset.$$

Thus, it is enough to show that any two vertices $v_1, v_2 \in U(T_1, T_2) = P'(T_1 \cup T_2)$ are connected by a path of length $O(\log n)$, in the subgraph induced by $U(T_1, T_2)$. We prove it by showing that the graph $H(T_1, T_2)$, induced by $U(T_1, T_2)$, is an expander.

We again use relation (1). In the original graph G, no set of vertices S, $|S| = \theta n$, had more than $d\theta^2 n/2 + \sqrt{d-1}\theta(1 - \theta)n$ internal edges (edges connecting vertices in S). Consider a set $S \subset U(T_1, T_2)$, $|S| = \theta n \leq n/2$. The degree of each vertex in $H(T_1, T_2)$ is at least $3d/5$. Thus, the number of edges connecting vertices in S to vertices in $U(T_1, T_2) \setminus S$ is at least

$$3d\theta n/5 - 2d\theta^2 n/2 - 2\sqrt{d-1}\theta(1 - \theta)n$$

$$\geq \theta nd/20,$$

for sufficiently large (constant) d. Dividing by the maximal degree in $H(T_1, T_2)$, we conclude that the set S is connected in $H(T_1, T_2)$ to at least $|S|/20$ vertices outside S, or $H(T_1, T_2)$ is an expander graph. □

Conclusion of the proof of Theorem 2: We use the same algorithm as in the proof of theorem 1 to obtain reliable communication between pairs of nodes in $P(T)$. However, messages are sent only on paths of length up to $D(G)$, were

$$D(G) = \max\{diameter(H(T_1, T_2)) \mid$$

$$T_i \subset V,\ |T_i| \leq \alpha|V|,\ i = 1, 2\}.$$

Since the graph $H(T_1, T_2)$ is always an expander, $D(G) = O(\log n)$. Thus, the algorithm requires only $O(\log n)$ communication rounds, and it generates polynomial number of messages.

Assume that T_1 is the set of faulty processors, $|T_1| \leq \alpha n$, $u, v \in P(T_1)$, and v sends messages to u. As in the proof of theorem 1, u tries to extract the correct value from the set of messages it receives from v by trying possible sets of faulty processors. When u ignores all messages that traverse the set T_1, it receives a consistent set of messages from v, and it deduces the correct value. Our construction guarantees that if u tries any other set T_2, there is a path of length no larger than $D(G)$ that connects v to u and does not traverse vertices in $T_1 \cup T_2$. Thus, when ignoring messages that traverse the set T_2, u receives at least one correct message, and the faulty processors can at most create an inconsistent set of values, from which u extract nothing. □

Corollary 1 *There is a constant $\alpha > 0$ and an n-vertex bounded degree network G, that can be explicitly constructed, such that G admits a.e.-agreement for up to αn faulty nodes.*

Proof: Theorem 2 proves that there is a bounded degree network G and a communication protocol P, such that for any set of $t \leq \alpha n$ faults, P guarantees reliable communication between all pair of processors in a set of at least $n - \mu t$ non-faulty processors.

Let PB be an agreement protocol for a complete network with up to μt faulty processors. Simulating the protocol PB on the network G using the communication protocol P guarantees agreement among at least $n - \mu t$ non-faulty nodes in the presence of up to $t \leq \alpha n$ faulty nodes. □

Acknowledgements:
I wish to thank N. Alon for helpful discussions, and for referring me to [1].

References

[1] N. Alon and F.R.K. Chung. Explicit construction of linear sized tolerant networks. *Discrete Mathematics*, 72:15–19, 1989.

[2] M. Ben-Or and D. Goldriech, Agreement in the presence of faults on networks of constant degree. Submitted, 1992.

[3] P. Berman and J.A. Garay, Fast Consensus in Networks of Bounded Degree. *4th International Workshop on Distributed Algorithms*, LNCS 486 (Springer-Verlag), pp.321-333, September 1990.

[4] P. Berman and J.A. Garay, Asymptotically Optimal Distributed Consensus. *Proc. 16th ICALP*, LNCS 372 (Springer-Verlag), pp. 80-94, July, 1989.

[5] Dolev D., The Byzantine generals strike again. *J. of Algorithms*, Vol. 3, No. 1 (1982), pp 14-30.

[6] Dolev D., Fischer M.J., Fowler R., Lynch N.A., Strong R. An Efficient Algorithm for Byzantine Agreement Without Authentication. *Information and Control* 52(3), pp.256-274 (1982).

[7] Dwork D., Peleg D., Pippenger N., Upfal E., Fault tolerance in networks of bounded degree. *SIAM J. Computing*, Vol. 17, (1988) pp. 975–988.

[8] Dana Goldriech (Ron), Communication In the Presence of Faults, On Networks of Bounded Degree. M.Sc. thesis, Department of Computer Science, The Hebrew University, Jerusalem, Israel, July 1989.

[9] Hadzilacos V., Issues of fault tolerance in concurrent computations, *Ph.D. Dissertation, Harvard University*, 1984.

[10] Lamport L., Shostak R., and Pease M., The Byzantine generals problem. *ACM Tran. on Prog. Lang. and Syst.*, Vol. 4, No. 3 (1982), pp. 382-401.

[11] Lubotzky A., Phillips R., and Sarnak P., Ramanugan Congecture and Explicit Constructions of Expanders. *18th Annual Symposium on Theory of Computing*, 1986, pp 240-246.

Performing Work Efficiently in the Presence of Faults

Cynthia Dwork
IBM Almaden Research Center
dwork@almaden.ibm.com

Joseph Y. Halpern
IBM Almaden Research Center
halpern@almaden.ibm.com

Orli Waarts
Stanford University
orli@cs.stanford.edu

Abstract: We consider a system of t synchronous processes that communicate only by sending messages to one another, and that together must perform n independent units of work. Processes may fail by crashing; we want to guarantee that in every execution of the protocol in which at least one process survives, all n units of work will be performed. We consider three parameters: the number of messages sent, the total number of units of work performed (including multiplicities), and time. We present three protocols for solving the problem. All three are work-optimal, doing $O(n+t)$ work. The first has moderate costs in the remaining two parameters, sending $O(t\sqrt{t})$ messages, and taking $O(n+t)$ time. This protocol can be easily modified to run in any completely asynchronous system equipped with a failure detection mechanism. The second sends only $O(t \log t)$ messages, but its running time is large ($O(t^2(n+t)2^{n+t})$). The third is essentially time-optimal in the (usual) case in which there are no failures, and its time complexity degrades gracefully as the number of failures increases.

1 Introduction

A fundamental issue in distributed computing is fault-tolerance: guaranteeing that work is performed, despite the presence of failures. For example, in controlling a nuclear reactor it may be crucial for a set of valves to be closed before fuel is added. Thus, the procedure for verifying that the valves are closed must be highly fault-tolerant. If processes never fail then the *work* of checking that the valves are closed could be distributed according to some load-balancing technique. Since processes may fail, we would like an algorithm that guarantees that the work will be performed as long as at least one process survives.

The notion of work in this paper is very broad, but is restricted to "idempotent" operations, that is, operations that can be repeated without harm. This is because if a process performs a unit of work and fails before telling a second process of its achievement, then the second process has no choice but to repeat the given unit of work. Examples include verifying a step in a formal proof, evaluating a boolean formula at a particular assignment to the variables, sensing the status of a valve, closing a valve, sending a message, say, to a process outside of the given system, printing a file, or reading records in a distributed database.

Formally, we assume that we have a synchronous system of t processes that are subject to crash failures, that want to perform n independent units of work. (For now, we assume that initially there is common knowledge among the t processes about the n units of work to be performed. We return to this point later.) Given that performing a unit of work can be repeated without harm, a trivial solution is obtained by having each process perform every unit of work. In our original example, this would mean that every process checks that every valve is closed. This solution requires no messages, but in the worst case performs tn units of work and runs in n rounds. (Here the worst case is when no process fails.)

Another straightforward solution can be obtained by having only one process performing the work at any time, and checkpointing to each pro-

The third author was supported by U.S. Army Research Office Grant DAAL-03-91-G-0102, NSF grant CCR-8814921, ONR contract N00014-88-K-0166, and an IBM fellowship.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0091...$1.50

cess after completing every unit of work. In this solution, at most $n + t - 1$ units of work are ever performed, but the number of messages sent is almost tn in the worst case.

In both these solutions the total amount of *effort*, defined as work plus messages, is $O(tn)$. If the actual cost of performing a unit of work is comparable to the cost of sending a message, then neither solution is appealing. In this abstract we focus on solutions which are work-optimal, up to a constant factor, while keeping the total effort reasonable. Clearly, since a process can fail immediately after performing a unit of work, before reporting that unit to any other process, a work-optimal solution performs $n + t - 1$ units of work in the worst case. Thus, we are interested in solutions that perform $O(n + t)$ work.

Let $n' = \max(n, t)$. Our first result is an algorithm whose total effort is at most $3n' + 9t\sqrt{t}$. In fact, in the worst case the amount of work performed is at most $3n'$ and the number of messages is at most $9t\sqrt{t}$, so the form of the bound explains the costs exactly. We then optimize this algorithm to achieve running time of $O(n + t)$ rounds. Note that any solution requires n rounds in the worst case, since if $t - 1$ processes are initially faulty then the remaining process must perform all n units of work. In this algorithm the synchrony is used only to detect failures, as usual by detecting the absence of an expected message. Thus, it can be easily modified to work in a completely asynchronous system equipped with a failure detection mechanism.

We then prove that the above algorithm is not message-optimal (among work-optimal algorithms), by constructing a technically challenging work-optimal algorithm that requires only $O(t \log t)$ messages in the worst case. Since $O(n+t)$ is a lower bound on work, and hence on effort, the $O(n + t \log t)$ effort of this algorithm is nearly optimal. The improved message complexity is obtained by a more subtle use of synchrony. In particular, the absence of a message in this algorithm has two possible meanings: either the potential sender failed or it has insufficient "information" (generally about the history of the execution), and therefore has chosen not to send a message. Due to this use of synchrony, unlike the first algorithm, this low-effort algorithm will not run in the asynchronous model with failure detection. In addition, the efficiency comes at a price in time: the algorithm requires $O(t^2(n + t)2^{n+t})$ rounds in the worst case.

The first two algorithms are very sequential: at all times work is performed by a single active process who uses some checkpointing strategy to inform other processes about the completed work. This forces the algorithms to take at least n steps, even in a failure-free run. To reduce the time we need to increase parallelism. However, intuitively, increasing parallelism while simultaneously minimizing time and remaining work-optimal may increase communication costs, since processes must quickly tell each other about completed work. The third algorithm does exactly this in a fairly straightforward way, paying a price in messages in order to decrease best-case time. It is designed to perform time-optimally in the absence of failures, and to have its time complexity degrade gracefully with additional faults. In particular, it takes $n/t + 2$ rounds in the failure-free case, where its message cost is $O(t)$; its worst-case message cost is $O(ft^2)$, where f is the actual number of failures in the execution. We postpone further discussion of this algorithm to the full paper.

One application of our algorithms is to Byzantine agreement. The idea is that the general tries to inform t processes, and then each of these t processes performs the "work" of ensuring that all processes are informed. In particular, our $O(t \log t)$-message solution yields an agreement algorithm for the crash fault model that requires fewer messages than any other algorithm in the literature. The best previous result is a nonconstructive algorithm due to Bracha that requires $O(n + t\sqrt{t})$ messages, where n is the total number of processes in the system, and t is a bound on the number of failures [4].

Using the observation that our solutions to the work problem yield solutions to Byzantine agreement, we can now return to the assumption that initially there is common knowledge about the work to be performed. Specifically, if even one process knows about this work, then it can act as a general, run Byzantine agreement on the pool of work using one of the three algorithms, and then the actual work is performed by running the same algorithm a second time on the real work. If n, the amount of actual work, is $\Omega(t)$, then the overall cost at most doubles when the work is not initially common knowledge.

The idea of doing work in the presence of failures, in a different context, has appeared elsewhere. In a seminal paper ([5]) Kanellakis and Shvartsman consider the *Write-All* problem, in which a set of n processes cooperates to set all n entries of an n-element array to the value 1. They provide an

efficient solution that tolerates up to $n-1$ faults, and show how to use it to derive robust versions of parallel algorithms for a large class of interesting problems. The paper was followed by a number of papers that consider the problem in other shared memory models (see [3, 6, 7, 8, 9]).

The Write-All problem is, of course, a special case of the type of work we consider. Nevertheless, our framework differs from that of [5] in two important respects, so that their results do not apply to our problem (nor ours to theirs). First, they consider the shared memory model while we consider the message passing model. Using the shared memory model simplifies things considerably for our problem. In the shared memory model, there is a straightforward algorithm (that uses shared memory to record what work has been done) with optimal effort $O(n+t)$, running in time $O(nt+t^2)$. While there are well-known emulators that can translate algorithms from the shared memory model to the message passing model (see [1, 2]), these emulators are not applicable for our problem, because the number of failures they tolerate is less than a majority of the total number of processes, while our problem allows up to $t-1$ failures. Also, these transformations introduce a multiplicative overhead of message complexity that is polynomial in t, while one of our goals here is to minimize this term.[1] Second, our complexity measure is inherently different from that of [5]. Kanellakis and Shvartsman's complexity measure is the sum, over the rounds during which the algorithm is running, of the number of processes that are not faulty during each round. This measure essentially "charges" for a nonfaulty process at round r whether it is actually doing any work (say, reading or writing a cell in shared memory), or not. Our approach is generally *not* to charge a process in round r if it is not expending any effort (sending a message or performing a unit of work) at that round, since it is free at that round to be working on some other task.[2]

[1] In fact, these emulators are designed for asynchronous systems, and hence it may be possible to improve their resilience for our synchronous model. Nevertheless, a multiplicative overhead in message complexity that is at least linear in t seems to be inherent in them.

[2] Inactive processes in our algorithms may need to both receive messages and count the number of rounds that have passed, say from the time they received their last message. We assume that processes can do this while carrying on other tasks.

2 A Protocol with Effort $O(n+t^{3/2})$

Our goal in this section is to present a protocol with effort $O(n+t\sqrt{t})$ and running time $O(n+t)$. We begin with a protocol that is somewhat simpler to present and analyze, with effort $O(n+t\sqrt{t})$ and running time $O(nt+t^2)$. This protocol has the additional property of working with minimal change in an asynchronous environment with failure detection.

The main idea of the protocol is to use checkpointing in order to avoid redoing too much work if a process fails. The most naïve approach to checkpointing does not work. To understand why, suppose a process does a checkpoint after each n/k units of work. This means that up to n/k units of work are lost when a process fails. Since up to t processes may fail, this means that nt/k units of work can be lost (and thus must be repeated), which suggests we should take $k \geq t$ if we want to do no more than $O(n)$ units of work altogether. However, since each checkpoint involves t messages, this means that roughly tk messages will be sent. Thus, we must have $k \leq \sqrt{t}$ if we are to use fewer than $t\sqrt{t}$ messages. Roughly speaking, this argument shows that doing checkpoints too infrequently means that there might be a great deal of wasted work, while doing them too often means that there will be a great deal of message overhead. Our protocol avoids these problems by doing *full* checkpoints to all the processes relatively infrequently—after $n/\sqrt{t}$ units of work—but doing partial checkpoints to only $\sqrt{t}$ processes after every n/t units of work. This turns out to be just the right compromise.

2.1 Description of the Algorithm

For ease of exposition, we assume that t is a perfect square, and that n is divisible by t (so that, in particular, $n > t$). We leave to the reader the easy modifications of the protocol when these assumptions do not hold. We assume that the processes are numbered 0 through $t-1$, and that the units of work are numbered 1 through n. We divide the processes into $\sqrt{t}$ groups of size $\sqrt{t}$ each, and use the notation g_i to denote process i's group. (Note $g_i = \lceil (i+1)/\sqrt{t} \rceil$.) We divide the work into $\sqrt{t}$ *chunks*, each of size $n/\sqrt{t}$, and subdivide the chunks into $\sqrt{t}$ *subchunks* of size n/t.

The protocol guarantees that at each round, at most one process is *active*. The active process is the only process performing work. If process i is active, then it knows that processes 0 to $i-1$ have crashed

or terminated. Initially, process 0 is active. The algorithm for process 0 is straightforward: Process 0 starts out doing the work, a subchunk at a time. After completing a subchunk c, it does a checkpoint to the remaining processes in its group g_0 (processes 1 to $\sqrt{t}-1$); that is, it informs its group that the subchunk of work has been completed by broadcasting to the processes in its group a message of the form (c, g_0). (If process 0 crashes in the middle of a broadcast, we assume only that some subset of the processes receive the message.) We call this a *partial checkpoint*, since the checkpointing is only to the processes in g_0. After completing a whole chunk of work—that is, after completing a subchunk c which is a multiple of $\sqrt{t}$—process 0 informs *all* the processes that chunk c has been completed, but it informs them one group at a time. After informing a whole group, it checkpoints the fact that a group has been informed to its own group (i.e., group 1). Formally, after completing a chunk c that is a multiple of $\sqrt{t}$, process 0 does a partial checkpoint to its own group, and then for each group $2, \ldots, \sqrt{t}$, process 0 broadcasts to the processes in group g a message of the form (c, g), and then broadcasts to all the processes in its own group a message of the form (c, g). We call this a *full checkpoint*. Note that in a full checkpoint, there is really a double checkpointing process: we checkpoint both the fact that work has been completed, and (to the processes in g_0) the fact that all processes have been informed that the work has been completed. Process 0 terminates after sending the message $(t, \sqrt{t})$ to process $t-1$, indicating to the last process that the last chunk of work has been completed (unless it crashes before that round).

If process 0 crashes, we want process 1 to become active; if process 1 crashes, we want process 2 to become active, and so on. More generally, if process j discovers that the first $j-1$ processes have crashed, then it becomes active. Once process j becomes active, it continues with essentially the same algorithm as process 0, except that it does not repeat the work it knows has already been done. We must ensure that the takeover proceeds in a "smooth" manner, so that there is at most one active process at a time.

Process j's algorithm is as follows. If j does not know that all the work has already been performed and sufficiently long time has passed from the beginning of the execution, then j becomes active. "Sufficiently long" means long enough to ensure that processes $0, \ldots, j-1$ have crashed or terminated. As we show below, we can take "sufficiently long" to be defined by the function $DD(j) = j(n+3t)$. ("DD" stands for deadline. We remark that this is not an optimal choice for the deadline; we return to this issue later.) Thus, if the round number r is $< DD(j)$, then j does nothing. Otherwise, if j does not know that the work is completed, it takes over as the active process at round $DD(j)$.

When j takes over as the active process, it essentially repeats process 0's algorithm. Suppose the last message j received was of the form (c, g). Then j starts by checkpointing the fact that it is now active to the remaining processes in its own group g_j (those processes with numbers higher than j, since the remainder are known to have crashed or terminated), by broadcasting the message (c, g) to them. Next there are now two cases. If c is not a multiple of $\sqrt{t}$, then j continues with the work in subchunk $c+1$. Process j does a partial checkpoint after completing each subchunk d, informing the remaining members of its group that d has been completed by broadcasting the message (d, g_j). If d marks the completion of a whole chunk of work, then process j performs a full checkpoint, informing all processes, a group at a time, by broadcasting the message (d, g) to group g, and checkpointing to the remaining members of its own group after completing the checkpoint to group g by broadcasting to them the message (d, g). If c is a multiple of $\sqrt{t}$, then j continues with the full checkpoint, starting with group $g+1$. That is, it broadcasts to each group $h = g+1, \ldots, \sqrt{t}$ the message (c, h), each time checkpointing its progress in the full checkpoint by broadcasting (c, h) to the remaining processes in g_j. Thus, if a process i receives a message of the form (c, g_i), it learns that subchunk c and all lower numbered subchunks have been completed. If it receives a message of the form (c, g) for $g \neq g_i$, then the sender of the message is in i's group, a full checkpoint is in progress, and group g has been informed that subchunk c and all lower numbered subchunks have been completed.

Process j terminates either upon receiving a message of the form (t, g) (since then it knows that all the work has been completed) or after sending the message $(t, \sqrt{t})$ to process $t-1$ (unless it crashes before that round). (Of course, if process $t-1$ becomes active, it terminates after completing all the work, since it never has to send checkpointing messages.) This completes the description of our first protocol. We call this Protocol $\mathcal{A}$.

Notice that we can easily modify this algorithm to run in a completely asynchronous system with a failure detection mechanism. We assume that, if someone fails, then the failure detection mechanism will eventually inform all the processes that have not failed of this fact. The modification is trivial: rather than waiting until round $DD(i)$ before becoming active, process i waits until it has been informed that processes $1, \ldots, i-1$ crashed or terminated.

2.2 Analysis and Proof of Correctness

We now give a fairly complete correctness proof for this protocol, to give the reader an idea of the type of arguments that need to be made. (Full proofs of correctness of our protocols are omitted for lack of space.) We say a process is *retired* if it has either crashed or terminated.

Lemma 2.1 *A process performs at most n units of work, sends at most $3t\sqrt{t}$ messages, and runs for less than $n + 3t$ rounds from the time it becomes active to the time it retires.* ∎

The following lemma is now immediate from the definition of DD.

Lemma 2.2 *Assume process i becomes active at round r of an execution $e_{\mathcal{A}}$ of protocol $\mathcal{A}$. Then all processes $< i$ have retired before round r.* ∎

In the sequel, it will sometimes be convenient to view a group g_i as a whole. Therefore we say that a *group is active* in the period starting when some process in this group becomes active and ending when the last process of this group retires. Notice that Lemma 2.2 ensures that when g_i becomes active, all processes in smaller groups have retired.

Theorem 2.1 *In every execution of protocol $\mathcal{A}$,*

(a) no more than $3n$ units of work are performed in total by the processes;

(b) no more than $9t\sqrt{t}$ messages are sent; and

(c) by round $nt + 3t^2$, all processes have retired.

Proof: Part (c) is immediate from Lemma 2.2 and the definition of DD.

We prove parts (a) and (b) simultaneously. To do so, we need a careful way of counting the total number of messages sent and the total amount of work done. A given unit of work may be performed a number of times. If it is performed more than once, say by processes $i_1, \ldots, i_k$, we say that i_2 *redoes* that unit of work of i_1, i_3 redoes the work of i_2, etc. It is important to note that i_3 does not redo the work of i_1 in this case; only that of i_2. Similarly, we can talk about a message sent during a partial checkpoint of a subchunk or a full checkpoint of a chunk done by i_1 as being *resent* by i_2.

Since the completion of a chunk is followed by a full checkpoint, it is not hard to show that when a new group becomes active, it will redo at most one chunk of work that was already done by previous active groups. It will also redo at most one full checkpoint that was done already on the previous chunk, and $\sqrt{t}$ partial checkpoints (one for each subchunk of work redone). Finally, if $g_j < g_i$, and the last message sent by process j before crashing is a broadcast to process i's group that was not received by i, process i must resend this broadcast. In all, it is easy to see that at most $n/\sqrt{t}$ units of work done by previous groups are redone when a new group becomes active, and $3t$ messages are resent. Similarly, since the completion of a subchunk is followed by a partial checkpoint, it is not hard to show that when a new process, say i, in a group that is already active becomes active, and the last message it received was of the form (c, g_i) (i.e., a partial checkpoint of subchunk c), it will redo at most one subchunk that was already done by previous active process (namely, $c+1$), and may possibly resend the messages in two partial checkpoints: the one sent after subchunk c, and the one sent after subchunk $c+1$ (if the previous process crashed during the checkpointing of $c+1$ without i receiving the message). If the last message that i received was (c, g) for $g > g_i$ (that is, the checkpointing of a checkpoint in the middle of a full checkpoint), then similar arguments show that it may resend $3\sqrt{t}$ messages: the checkpoint of (c, g) to its own group, the checkpoint $(c, g+1)$ to group $g+1$, and the checkpointing of $(c, g+1)$ to its own group. Thus, the amount of work done by an active group that is redone when a new process in that group becomes active is at most n/t, and the number of messages resent is at most $3\sqrt{t}$.

The maximum amount of unnecessary work done is: (number of groups) × (amount of work redone when a new group becomes active) + (number of processes) × (amount of work redone when a new process in an already active group becomes active) $\leq \sqrt{t}(n/\sqrt{t}) + t(n/t) = 2n$. Similarly, the maximum

number of unnecessary messages that may be sent is no more than: (number of groups) × (number of messages resent when a new group becomes active) + (number of processes) × (number of mesages sent when a new process in an already active group becomes active) $\leq \sqrt{t}(3t) + t3\sqrt{t} = 6t\sqrt{t}$. Clearly n units of work must be done; by Lemma 2.1, at most $3t\sqrt{t}$ messages must be sent. Thus, no more than $3n$ units of work will be done altogether, and no more than $9t\sqrt{t}$ messages will be sent altogether. ∎

2.3 Improving the Time Complexity

As we have observed, the round complexity of Protocol $\mathcal{A}$ is $nt + 3t^2$. We now discuss how the protocol can be modified to give a protocol that has round complexity $O(n + t)$, while not significantly changing the amount of work done or the number of messages sent.

Certainly one obvious hope for improvement is to use a better function than DD for computing when process i should become active. While some improvement is possible by doing this, we can get a round complexity of no better than $O(n\sqrt{t})$ if this is all we do, which is still more than we want. Intuitively, the problem is that if process j gets a message of the form (c, g), where c is a multiple of $\sqrt{t}$, then it is possible, as far as j is concerned, that some other process $i < j$ may have received a message of the form $(c + \sqrt{t}, h)$. Process j cannot become active before it is sure that i has retired. To compute how long it must wait before becoming active, it thus needs to compute how long i would wait before becoming active, given that i got a message of the form $(c + \sqrt{t}, h)$. On the other hand, if i did get such a message, then as far as i is concerned, some process $i' < i$ may have received a message of the form $(c + 2\sqrt{t}, h')$. Notice that, in this case, process j knows perfectly well that no process received a message of the form $(c+2\sqrt{t}, h')$; the problem is that i does not know this, and must take into account this possibility when it computes how long to wait before becoming active. Carrying out a computation based on these arguments gives an algorithm which runs in $O(n\sqrt{t})$ rounds.

On closer inspection, it turns out that the situation described above really causes difficulties only when all processes involved (in the example above, this would be the processes j, i, and i') are in the same group. Thus, in our modified algorithm, process j computes the time to become active as follows: Suppose that the last message received by process j before round r is (c, g), and this message was received from process i at round r'. Process j then computes a function $F(j, c, g, i)$ with the property that if $r = r' + F(j, c, g, i)$, then process j knows at round r that all processes in groups $g' < g_j$ must have retired. (If $g_i = g_j$ then $F(j, c, g, i) = 0$.) Process j then polls all the lower-numbered processes in its own group, one by one, to see if they are alive; if not, then j becomes active. If any of them is alive, then the lowest-numbered one that is alive becomes active upon receiving j's message. Once a process becomes active, it proceeds just as in Protocol $\mathcal{A}$. This technique turns out to save a great deal of time, while costing relatively little in the way of messages. We leave details of the computation of $F(j, c, g, i)$ and the correctness of this protocol to the full paper.

3 An Algorithm with Effort $O(n + t \log t)$

In this section we prove that the effort of $O(n+t\sqrt{t})$ obtained by the previous protocols is not optimal, even for work-optimal protocols. We construct another work-optimal algorithm, Protocol $\mathcal{C}$, that requires only $O(n + t \log t)$ messages (and a variant that requires only $O(t \log t)$ messages), yielding a total effort of $O(n + t \log t)$. As is the case with protocols $\mathcal{A}$ and $\mathcal{B}$, at most one process is active at any given time. However, in protocol $\mathcal{C}$ it is not the case that there is a predetermined order in which the processes become active. Rather, when an active process fails, we want the process that is currently ***most knowledgeable*** to become the new active process. As we shall see, which process is most knowledgeable after an active process i fails depends on how many units of work i performed before failing. As a consequence, there is no obvious variant of protocol $\mathcal{C}$ that works in the model with asynchronous processes and a failure-detector.

Roughly speaking, Protocol C strives to "spread out" as uniformly as possible the knowledge of work that has been performed and the processes that have crashed. Thus, each time the active process, say i, performs a new unit of work or detects a failure, i tells this to the process j it currently considers least knowledgeable. Then process j becomes as knowledgeable as i, so after performing the next unit of work (or detecting another failure), i tells the process it now considers least knowledgeable about this new fact.

The most naïve implementation of this idea is

the following: Process 0 begins by performing unit 1 of work and reporting this to process 1. It then performs unit 2 and reports units 1 and 2 to process 2, and so on, telling process i mod t about units 1 through i. Note that at all times, every process knows about all but at most the last t units of work to be performed.

If process 0 crashes, we want the most knowledgeable alive process—the one that knows about the most units of work that have been done—to become active. (If no process alive knows about any work, then we want the highest numbered alive process to become active.) It can be shown that this can be arranged by setting appropriate deadlines. Moreover, the deadlines are chosen so that at most one process is active at a given time. The most knowledgeable process then continues to perform work, always informing the least knowledgeable process.

The problem with this naïve algorithm is that it requires $O(n+t^2)$ work and $O(n+t^2)$ messages in the worst case. For example, suppose that process 0 performs the first $t-1$ units of work, so that the last process to be informed is process $t-1$, and then crashes. In addition, $t/2+1,\ldots,t-1$ crash. Eventually process $t/2$, the most knowledgeable non-retired process, will become active. However, process $t/2$ has no way of knowing whether process 0 crashed just after informing it about work unit $t/2$, or process 0 continued to work, informing later processes (who must have crashed, for otherwise they would have become active before process $t/2$). Thus, process $t/2$ repeats work units $t/2+1,\ldots,t-1$, again informing (retired) processes $t/2+1,\ldots t-1$. Suppose process $t/2$ crashes after performing work unit $t-1$ and informing process $t-1$. Then process $t/2-1$ becomes active, and again repeats this work. If each process $t/2-1, t/2-2,\ldots,1$, crashes after repeating work units $t/2+1,\ldots,t-1$, then $O(t^2)$ work is done, and $O(t^2)$ messages are sent. (A slight variant of this example gives a scenario in which $O(n+t^2)$ work is done, and $O(n+t^2)$ messages are sent.)

To prevent this situation, a process performs failure detection before proceeding with the work. The key idea here is that we treat failure detection as another type of work. This allows us to use our algorithm recursively for failure detection. Specifically, fault-detection is accomplished by polling a process and waiting for a response or a timeout. The difficulty encountered by our approach is that, in contrast to the real work, the set of faulty processes is dynamic, so it is not obvious how these processes can be detected without sending (wasteful) polling messages to *nonfaulty* processes. In fact, in our algorithm we do not attempt to detect all the faulty processes, only enough to ensure that not too much work is wasted by reporting work to faulty processes.

3.1 Description of the Algorithm

For ease of exposition we assume t is a power of 2. Again, the processes are numbered 0 through $t-1$, and the units of work are numbered 1 through n. Although our algorithm is recursive in nature, it can more easily be described when the recursion is unfolded. Processing is divided into $\log t$ levels, numbered 1 to $\log t$, where level $\log t$ would have been the deepest level of the recursion, had we presented the algorithm recursively. In each level, the processes are partitioned into groups as follows. In level h, $1 \le h \le \log t$, there are $t/(2^{\log t-h+1})$ groups of size $2^{\log t-h+1}$. Thus, in level $\log t$, there are $t/2$ groups of size 2, in level $\log t-1$ there are $t/4$ groups of size 4, and so on, until level 1, in which there is a single group of size t. Let $s_h = 2^{\log t-h+1}$ denote the size of a group at level h. The first group of level h contains processes $0,1,\ldots,s_h-1$, the next group contains processes $s_h, s_h+1,\ldots,2s_h-1$, and so on. Thus each group of level $h < \log t$ contains two groups of level $h+1$. Note that each process i belongs to $\log t$ groups, exactly one on each level. We let G_h^i denote the level h group of process i.

Initially process 0 is active. When process i becomes active, it performs fault-detection in its group at every level, beginning with the highest level and working its way down, leaving level h as soon as it finds a non-faulty process in G_h^i. Once fault-detection has been completed on G_1^i, the set of all processes, process i begins to perform real work. Thus, we sometimes refer to the actual work as G_0, or level 0, and the fault-detection on level h as *work on level h*. For each $1 \le h \le \log t$, each time it performs a unit of work on G_{h-1}^i, process i reports that work to some process in G_h^i.

A *unit of fault-detection* is performed by sending a special message "Are you alive?" to one process, and waiting for a reply in the following round. An *ordinary* message informs a process at some level h, $1 \le h \le \log t$, of a unit of (real or fault-detection) work at level $h-1$. As we shall see, an ordinary message also carries additional information. These two are the only types of messages sent by an active process. As before, a process that has crashed or terminated is said to be *retired*. An inactive non-

retired process only sends responses to "Are you alive?" messages.

Each process i maintains a list F_i of processes known by i to be retired. It also maintains an array of pointers, POINT_i, indexed by group name. Intuitively, $\text{POINT}_i[G_0]$ is the successor of the last unit of work known by i to have been performed (and therefore this is where i will start doing work when it becomes active). For $h \geq 1$, $\text{POINT}_i[G_h^j]$ contains the successor (according to the cyclic order in G_h^j) of the last process in G_h^j known by i to have received an ordinary message from a process in G_h^j that was performing (real or fault-detection) work on G_{h-1}^j. We call $\text{POINT}_i[G_h^j]$ *process i's pointer into* G_h^j. Process i's moves are governed entirely by the round number, F_i, and pointers into its own groups (i.e., pointers into groups G_h^i). Associated with each pointer $\text{POINT}_i[G]$ is a round number, $\text{ROUND}_i[G]$, indicating the round at which the last message known to be sent was sent (or, in the case of G_0, when the last unit of work known to be done was done). Initially, $\text{POINT}_i[G_0] = 1$, $\text{POINT}_i[G_h^j]$ is the lowest-numbered process in $G_h^j - \{i\}$, and $\text{ROUND}_i[G_0] = \text{ROUND}_i[G_h^j] = 0$. We occasionally use $\text{ROUND}_i[G](r)$ to denote the value of $\text{ROUND}_i[G]$ at the beginning of round r; we similarly use $F_i(r)$ and $\text{POINT}_i[G](r)$.

The triple $(F_i, \text{POINT}_i, \text{ROUND}_i)$ is the *view* of process i. We also define the *reduced view* of process i to be $\text{POINT}_i[G_0] - 1 + |F_i|$; thus, i's reduced view is the sum of the number of units of work known by i to be done and the number of processes known by i to be faulty. A process includes its view whenever it sends an ordinary message. When process i receives an ordinary message, it updates its view in light of the new information received. Note that process i may receive information about one of its own groups from a process not in that group. Similarly, it may pass to another process information about a group in which the other process is a member but to which i does not belong.

Let G_h^i be any group as described above, where the process numbers range from x to $y = x + |G_h^i| - 1$. There is a natural fixed cyclic order on the group, which we call the *cyclic order*. Process i sends messages to members of G_h^i in *increasing order*. By this we mean according to the cyclic order but skipping itself and all processes in F_i. Let $j \neq i$ be in G_h^i. Then j's *i-successor in G_h^i*, is j's nearest successor in the cyclic ordering that is not in $\{i\} \cup F_i$. We omit the i in "i-successor," as well as the name of the group in which the successor is to be determined, when these are clear from the context.

When process i first becomes active it searches for other non-retired processes as follows. For each level h, starting with $\log t$ and going down to 1, process i polls group G_h^i, starting with $\text{POINT}_i[G_h^i]$, by sending an "Are you alive?" message. If no answer is received, it adds this process to F_i. If $h < \log t$, process i sends an ordinary message reporting this newly detected failure to $\text{POINT}_i[G_{h+1}^i]$, sets $\text{POINT}_i[G_{h+1}^i]$ to its i-successor in G_{h+1}^i, and sets $\text{ROUND}_i[G_{h+1}^i]$ to the current round number. Process i repeats these steps until an answer is received or $G_h^i \setminus \{i\} \subseteq F_i$. It then enters level $h-1$, and repeats the process. Level 0 is handled similarly to levels 1 through $\log t - 1$, but the process performs real work instead of polling, and increases the work pointer after performing each unit of work. If $\text{POINT}_i[G_0] = n$ then process i halts, since in this case all the work has been completed. This completes the description of the behavior of an active process. The code for an active process appears in Figure 1.

At any time in the execution of the algorithm, each inactive non-retired process i has a *deadline*. We define $D(i,m)$ to be the number of rounds that process i waits from the round in which it first obtained reduced view m until it becomes active:

$$D(i,m) = \begin{cases} K(n+t-m)2^{n+t-1-m} & \text{if } m \geq 1 \\ K(t-i)(n+t)2^{n+t-1} & \text{otherwise.} \end{cases}$$

where $K = 5t + 2\log t$. As we show below (Lemma 3.2), K is an upper bound on the number of rounds that any process needs to wait before first hearing from the active process. (More formally, if j becomes active at round r and is still active K rounds later, then by the beginning of round $r+K$, all processes that are not retired will have received a message from j.) All our arguments below work without change if we replace K by any other bound on the number of rounds that a process needs to wait before first hearing from the active process. This observation will be useful later, when we consider a slight modification of protocol $\mathcal{C}$.

If process i receives no message by the end of $D(i,0)-1$, then it becomes active at the beginning of round $D(i,0)$. Otherwise, if at round r it receives a message based on which it obtains a reduced view of m, and if it receives no further messages by the end of round $r + D(i,m) - 1$, it becomes active at the beginning of round $r + D(i,m)$. This completes the description of the algorithm.

```
1. h := log t;
2. While h > 0 do:
3.       DONE := FALSE;
4.       While ¬DONE do:
5.             Send "Are you alive?" to POINT_i[G^i_h];
6.             If no response
7.                   then add POINT_i[G^i_h] to F_i;
9.                   If h ≠ log t
10.                        then send ordinary message to point_i[G^i_{h+1}];
11.                        ROUND_i[G^i_{h+1}] := current round;
12.                        POINT_i[G^i_{h+1}] := successor(POINT_i[G^i_{h+1}]);
13.                  If G^i_h − F_i ≠ {i}
14.                        then POINT_i[G^i_h] := successor(POINT_i[G^i_h]);
15.                        else DONE := TRUE
16.                  else (i.e., response received) DONE := TRUE;
17.      h := h − 1;

Process level 0 (real work):
18. While POINT_i[G_0] ≤ n do:
19. Perform work unit POINT_i[G_0];
20. If POINT_i[G_0] ≠ n then
21.      Send an ordinary message to POINT_i[G^i_1];
22.      ROUND_i[G^i_1] := current round;
23.      POINT_i[G^i_1] := successor(POINT_i[G^i_1]);
24.      POINT_i[G_0] := successor(POINT_i[G_0]);
```

Figure 1: Code for Active Process i in Protocol $\mathcal{C}$

3.2 Analysis and Proof of Correctness

Lemma 3.1 *In every execution of Protocol $\mathcal{C}$ in which there are no more than $t-1$ failures, the work is completed.* ∎

The next lemma shows that our choice of K has the properties mentioned above.

Lemma 3.2 *If j is active at round r, and is not retired by round $r + 5t + 2\log t$, then all processes that are not retired will receive a message from j before the beginning of $r + 5t + 2\log t$.* ∎

If i received its last ordinary message from j at round r, we call other processes that received an ordinary message from j after i did *first-generation processes* (implicitly, with respect to i, j, and r). If i did not yet receive any ordinary messages, then the first-generation processes (with respect to i and r) are those that received an ordinary message from a process with a number greater than i. We define kth generation processes inductively. If we have defined kth generation, then the $(k+1)$st generation are those processes that receive an ordinary message from a kth generation process. The *rank* of a process is the highest generation that it is in.

Lemma 3.3 *Let i receive its last ordinary message from j at round r, let m be the reduced view of i after receiving this message, and let ℓ be a kth rank process with respect to i, j, and r. Then, after ℓ receives its last ordinary message, its reduced view is at least $m + k$.* ∎

We say process i *knows more* than process j at round r if $F_i(r) \supseteq F_j(r)$ and for all groups G, $\text{ROUND}_i[G](r) \geq \text{ROUND}_j[G](r)$. Note that if equality holds everywhere then intuitively the two processes are equally knowledgeable. We first show that our algorithm has the property that for any two inactive non-retired processes, one of them is more knowledgeable than the other, unless they both know nothing; that is, the knowledge of two

non-retired processes is never incomparable. This is important so that the "most knowledgeable" process is well-defined. Moreover, the knowledge can be quantified by the reduced view. Process i knows more than inactive process j if and only if the reduced view of i is greater than the reduced view of j. Finally, the algorithm also ensures that the active process is at least as knowledgeable as any inactive non-retired process.

Lemma 3.4 *For every round r of the execution the following hold:*

(a) If process i received an ordinary message from process j at round $r' < r$, and i is inactive and has not retired by the beginning of round r, then at the beginning of round r, no processes other than j and processes in the kth generation with respect to i, j, and r', for some $k \geq 1$, know as much as i.

(b) Suppose process i received its last ordinary message at round r' (if i has received no ordinary messages then $r' = 0$), and m is i's reduced view after receiving this message. If i is not retired at the beginning of round $r = r' + D(i,m)$, and it receives no further ordinary messages before the beginning of round r, then at the beginning of round r no non-retired process knows more than i.

(c) At most one process is active in round r.

(d) At the beginning of round r, there is an asymmetric total order ("knows more than") on the non-zero knowledge of the inactive non-retired processes, and the active process knows at least as much as the most knowledgeable among these processes. Moreover, for any two non-retired processes i and j, i knows more than j if and only if the reduced view of i is greater than the reduced view of j. ∎

Lemma 3.5 *The running time of the algorithm is at most $tK(n+t)2^{n+t}$ rounds.*

Proof: If process i's reduced view is m and it does not receive a message within $D(i,m)$ steps, then it becomes active. Each message that i receives increases its reduced view. Thus, i becomes active in at most $D(i,0) + \cdots + D(i,n+t-1)$ rounds. Once it becomes active, arguments similar to those used in Lemma 3.2 show that it retires in at most $2n + 3t + 2\log t$ rounds. Thus, the running time of the algorithm is at most $D(1,0) + \cdots + D(1,n+t-1) + 2n + 3t + 2\log t \leq tK(n+t)2^{n+t}$ rounds. ∎

The next lemma shows that an active process i does not send messages to retired processes that, because they were more knowledgeable than i, should have become active before i did. These messages are avoided because during fault detection i discovers that these processes have retired.

Lemma 3.6 *If process i' gets an ordinary message at round r' from a process operating on group $G^{i'}_{h-1}$ and process i is active at the beginning of round $r > r'$ then:*

(a) if $\text{ROUND}_i[G^{i'}_h](r) \geq r'$, then all processes in the interval $[i', \text{POINT}_i[G^{i'}_h](r))$ in the cyclic order on $G^{i'}_h$ are either retired by the beginning of round r or receive an ordinary message in the interval $[r', \text{ROUND}_i[G^{i'}_h](r)]$ from a process operating on $G^{i'}_{h-1}$. (If $i' = \text{POINT}_i[G^{i'}_h](r)$, then all processes in $G^{i'}_h$ are either retired by the beginning of round r or receive a message in the interval $[r', \text{ROUND}_i[G^{i'}_h](r)]$ from a process operating on $G^{i'}_{h-1}$.) Moreover, either i's knowledge at the beginning of round r is greater than i''s knowledge at the end of r', or $i' \in F_i(r)$.

(b) if $\text{ROUND}_i[G^{i'}_h](r) < r'$, then all processes in the interval $[\text{POINT}_i[G^{i'}_h](r), i']$ in the cyclic order on $G^{i'}_h$ are either retired by the beginning of round r', or receive a message in the interval $(\text{ROUND}_i[G^{i'}_h](r), r']$ from a process operating on $G^{i'}_{h-1}$. Moreover, all the processes in this interval are retired by the beginning of round r, and if $G^i_h = G^{i'}_h$, then all these processes will be in F_i by the time i begins to operate on G^i_{h-1}. ∎

Observe that the algorithm treats 'are you alive?' messages as real work. Therefore, in the sequel, we will refer to these messages as work unless stated otherwise. On the other hand, the ordinary messages are still referred to as messages.

Using Lemma 3.6, we can show that indeed effort is not wasted:

Lemma 3.7 *The number of work units done and reported to G^i_h by group G^i_h when operating on group G^i_{h-1} is no more than $|G^i_h| + |G^i_{h-1}|$.*

Proof: Given i, h, and an execution e of protocol C, we consider the sequence of triples (x,y,z), with

one triple in the sequence for every time a process $x \in G_h^i$ sends an ordinary message reporting a unit of work $y \in G_{h-1}^i$ to a process $z \in G_h^i$, listed in the order that the work was performed. We must show that the length of this sequence is no more than $|G_{h-1}^i| + |G_h^i|$.

We say that a triple (x, y, z) is *repeated* in this sequence if there is a triple (x', y, z') later in the sequence where the same work unit y is performed. Clearly there are at most $|G_{h-1}^i|$ nonrepeated triples in the sequence, so it suffices to show that there are at most $|G_h^i|$ repeated triples. To show this, it suffices to show that the third components of repeated triples (denoting which process was informed about the unit of work) are distinct. Suppose, by way of contradiction, that there are two repeated triples (x_1, y_1, z_1) and (x_2, y_2, z_1) with the same third component. Suppose that x_1 informed z_1 about y_1 in round r', and x_2 informed z_1 about y_2 in round r''. Without loss of generality, we can assume that $r' < r''$. Since (x_1, y_1, z_1) is a repeated triple, there is a triple (x_3, y_1, z_2) after (x_1, y_1, z_1) in the sequence. Let r_3 be the round in which x_3 became active, and let r_2 be the round in which x_2 became active. Let $s_j = \text{ROUND}_{x_j}[G_h^i](r_j)$, for $j = 2, 3$. By Lemma 3.6, if $s_2 \geq r'$, then either x_2's knowledge at the beginning of round s_2 is greater than z_1's knowledge at the end of r', or $z_1 \in F_{x_2}(r')$, and if $s_2 < r'$, then $z_1 \in F_{x_2}$ before x_2 starts operating on G_i^{h-1}. Since x_2 sends a message to z_1 while operating on G_i^{h-1}, it cannot be the case that $z_1 \in F_{x_2}$ before x_2 starts operating on G_i^{h-1}, so it must be the case that $s_2 \geq r'$ and x_2's knowledge at the beginning of round r_2 is greater than z_1's knowledge at the end of round r'. In particular, this means that x_2 must know that x_1 informed z_1 about y_1 at the beginning of r_2.

We next show that every process $x \in G_i^h$ that is active at some round r between r' and r_2 must know that x_1 informed z_1 about y_1 at the beginning of round r. For suppose not. Then, by Lemma 3.6, z_1 must have retired by the beginning of round r. Since, by Lemma 3.4, x is the most knowledgeable process at the beginning of round r, it follows that no process that is not retired knows that z_1 was informed about y_1. Thus, there is no way that x_2 could find this out by round r_2.

It is easy to see that x_3 does not know that z_1 was informed about y_1 (for if it did, it would not repeat the unit of work y_1). Therefore, (x_3, y_1, z_2) must come after (x_2, y_2, z_1) in the sequence. Since $\text{POINT}_{x_2}[G_h^i](r'') = z_1$, and z_1 received an ordinary message from x_1 while operating on G_{h-1}^i at round r', it follows from Lemma 3.6 that between rounds r' and r'', every process in G_h^i that is not retired must receive an ordinary message. In particular, this means that x_3 must receive an ordinary message. Since all active processes between round r' and r'' know that z_1 was informed about y_1, it follows that x_3 must know it too by the end of round r''. But then x_3 would not redo y_1, giving us the desired contradiction. ∎

Theorem 3.1 *In every execution of Protocol C the following hold:*

(a) The total amount of real work performed is no more than $n + 2t$ units;

(b) The number of messages sent is no more than $n + 6t \log t + 4t$;

(c) The total number of rounds is no more than $t(5t + 2\log t)(n + t)2^{n+t}$.

Proof: Lemma 3.7 implies that the amount of real work units that are performed and reported to G_1 is no more than $|G_0| + |G_1| = n + t$. In addition, each of the t processes may perform one unit without reporting it (because it retired immediately afterwards). Summing the two, (a) follows.

Part (b) follows fairly easily from Lemma 3.7, while part (c) is immediate from Lemma 3.5. ∎

We remark that we can improve the message complexity to $O(t \log t)$ (that is, remove the n term in (b) above) by informing processes in group G_1 after n/t units of work done at level G_0, rather than after every unit of work. The total work done is still $O(n + t)$; the time complexity increases to $t(2n + 3t + 2\log t)(n + t)2^{n+t}$ because of an increase in K (the upper bound on the number of rounds, from the time the currently active process takes over, that any process needs to wait before first hearing from the active process).

4 Application to Byzantine Agreement

Each of our algorithms can be used to construct an algorithm for Byzantine agreement along the following lines. The general sends its value to processes $T = \{0, \ldots, t\}$ (note that at least one of these processes is non-faulty) and then decides on

this value. The $t+1$ processes then must perform the "work" of informing first the rest of T and then the remaining $n-t-1$ processes of the value heard from the general. Thus, the units of work are, in order, informing processes $1, 2, \ldots, n$. At all times a process' current value is the last value it has heard (0 if it has never heard anything). When a process becomes active, it informs others of its current value. At the end of the algorithm it decides on its current value.

The proof of correctness of this algorithm varies according to which of Algorithms $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ is used for performing work. In the first two cases the proof relies on the fact that processes are informed about work (more or less) in the same order in which the work was performed. In the last case the proof depends on the fact that the active process is always the most knowledgeable one.

We remark that not every algorithm for performing work yields an algorithm for Byzantine agreement along the lines that we have described (consider, for example, the trivial algorithm for performing work, in which all processes perform all n units of work).

5 Conclusions

In this paper we have formulated the problem of performing work efficiently in the presence of faults. We presented three work-optimal protocols to solve the problem. One sends $O(t\sqrt{t})$ messages and takes $O(n+t)$ time, another requires $O(t \log t)$ messages at the cost of significantly greater running time. In the full paper we present an algorithm that optimizes on time in the usual case (where there are few failures). In particular, in the failure-free case, it takes $n/t+2$ rounds and requires $O(t)$ messages. Its time performance degrades gracefully with additional failures, and its worst-case message complexity is $O(ft^2)$, where f is the actual number of faults in the execution.

It would be interesting to see if message complexity and running time could be simultaneously optimized. It would also be interesting to prove a nontrivial lower bound on the message complexity of work-optimal protocols.

Acknowledgements The authors are grateful to Vaughan Pratt for many helpful conversations, in particular for his help with the proof of Algorithm $\mathcal{A}$. We also thank Maurice Herlihy for his suggestions for improving the presentation of this work.

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 363-375, 1990.

[2] A. Bar-Noy and D. Dolev, "Shared-Memory vs. Message-Passing in an Asynchronous Distributed Enviroinment," *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 307-318, 1989.

[3] J. Buss and P. Ragde, "Certified Write-All on a Strongly Asynchronous PRAM," manuscript, 1990.

[4] G. Bracha, unpublished manuscript, Department of Computer Science, Cornell University, July 1984.

[5] P. Kanellakis and A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust," *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 211–219, 1989.

[6] P. Kanellakis and A. Shvartsman, "Efficient Parallel Algorithms on Restartable Fail-Stop Processes," *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pp. 23–35, 1991.

[7] Z. Kedem, K. Palem, A. Raghunathan, and P. G. Spirakis, "Combining Tentative and Definite Executions for Very Fast Dependable Parallel Computing," *Proc. 23rd ACM Symposium on Theory of Computing*, pp. 381–389, 1991.

[8] Z. M. Kedem, K. V. Palem, and P. G. Spirakis, "Efficient Robust Parallel Computations," *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138-148, 1990.

[9] C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pp. 590–599, 1991.

On the Complexity of Global Computation in the Presence of Link Failures: The Case of Uni-Directional faults *

Oded Goldreich and Dror Sneh
Computer Science Dept.
Technion, Haifa, Israel.

ABSTRACT

We consider distributed computations in an asynchronous communication model with undetectable link failures. The computational tasks we consider are obtaining the value of a predetermined function of the local inputs scattered in the network (e.g., the sum of all local values). We call this task Global Computation.

A trivial protocol for Global Computation consists of each processor sending its local input to all processors via flooding. Our aim is to justify the use of this simple protocol, in the presence of faulty links, by proving matching lower bounds on the message complexity (i.e., total number of messages sent) of Global Computation.

In this paper we concentrate on the case in which the communication links are either uni-directional or fail in a uni-directional manner. Our main result states that for every n and m, the message complexity of Global Computation on such networks is at least

$$\frac{n \cdot m}{PolyLog(n)}$$

where n is the number of processors and m is the number of links. Hence, in the presence of uni-directional link failures, the simple flooding algorithm is optimal up to a polylogarithmic factor.

* Partially supported by the Technion's Vice President for Research and Development grant.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0103...$1.50

1. INTRODUCTION

We consider distributed systems consisting of a set of processors and a set of links connecting pairs of processors. A basic task in a distributed system is computing a predetermined function of local inputs scattered in the system. For example, one may be interested in computing the sum of all local inputs. In general, one is interested in computing $f(x_1,...,x_n)$, where f is a predetermined function and x_i is the local input of processor i. The result of the computation should be known to all (or one of) the processors in the network. In general, computing the value of f may require knowledge of all the local inputs. The question is what is the cost, specifically the message complexity (i.e., the number of messages sent), of obtaining this knowledge. The answer, of course, depends on the computational model.

The natural computation models, for the above task, vary by the quality/reliability of the links connecting the various processors. We believe that these different models reflect different "levels of abstraction" applied to practical networks. A high-level model may consider communication networks as supporting synchronous communication. A more low-level model assumes only asynchronous message transmissions (but allows no faults). Assuming asynchronous communications with detectable faults is even more low-level, and waiving the assumption that faults are detectable goes even further. It should be stressed that all levels of abstraction are justified in some sense although none captures reality. Yet, one should remember that these different levels of abstraction correspond to different layers of communication protocols operating in the network (e.g., ISO layers) and that high levels of abstraction are obtained at the cost of more complex or expensive protocols.

In this paper we consider a very weak model of communication. Specifically, we consider an asynchronous (message passing) network with undetectable (fail-stop) link failures [IR,SR,GS]. The number of link failures is not a-priori bounded, except that it is guaranteed that these failures never disconnect the network.

A trivial protocol for computing the value of f consists of each processor sending its local input to all other processors in the network (by using a flooding protocol). Of course, for a specific "degenerated" function f (e.g., a constant function) much better protocols are possible. However, we are interested in alternative (and possibly cheaper) protocols for more typical functions and specifically for functions which are "sensitive to all their inputs" (e.g., SUM, AND, MAX etc.). Loosely speaking, our main result is that *in an asynchronous model of uni-directional links with undetectable faults no significant saving in complexity is possible.* A more precise statement follows.

We show that there exists a polynomial P so that for every n and m the following holds. Let Π be an arbitrary protocol, for computing the sum (or any other input-sensitive function) of the local values residing in the processors of some network with n processors and m uni-directional links (which may fail-stop in an undetectable manner). Then there exists an execution of Π, on such a network, in which at least $\frac{n \cdot m}{P(\log n)}$ messages are sent. We stress that our lower bound holds also in case only faulty links behave in a uni-directional manner. It should be stressed that the trivial protocol mentioned above can compute any function of the local inputs using $n \cdot m$ messages.

Of course, we would have been more happy to prove the above result in a model in which both faulty and non-faulty links are bi-directional. Yet, the uni-directional model of faults is well motivated. Even in a setting in which it is reasonable to assume that the initial topology of the network is "bi-directional" it is sometimes natural to postulate that the faults occurring in one direction of a link are "independent" of the performance of the other direction. Hence, the faults may be uni-directional even if non-faulty links are bi-directional. Furthermore, we hope that some of the ideas presented in the proof of the lower bound would be useful also for proving an analogous lower bound in the bi-directional fault model.

2. THE MODEL

Our model of computation is an *asynchronous* model of *uni-directional* communication with *undetectable link failures.* Namely, we consider an arbitrary directed graph with processors placed at the nodes and directed edges representing uni-directional communication links. Links are directed from their *tail* to their *head.* Each processor runs a predetermined local program. An assignment of local programs to all processors of the network is called a *protocol* (or an *algorithm*). An execution of a protocol on the above network is determined both by the protocol (and the initial local inputs) and by a scheduling of events agreeing with the standard link axioms. (The scheduling determines which of the receive-message events that may occur in a processor will occur first. A different scheduling of the receive-message events may cause the processor to behave differently. Specifically, this may cause different send-message events at the processor.) Loosely speaking, the link axioms determine that receive-message events occurring at the head of a link must be preceded by a "corresponding" send-message event occurring at the tail of this link. The correspondence of receive-message and send-message events over each link is a one-to-one (but not necessarily onto) function mapping receive-message events at the head of the link to send-message events at the tail of the link. Furthermore, this function must be order preserving (i.e., if a receive-message event r_1 precedes a receive-message event r_2 at the head of the link then the send-message event corresponding to r_1 must precede the send-message event corresponding to r_2).

A link is said to be *faulty* during an execution if the execution terminates and there exists a send-message event at the tail of this link without a corresponding receive-message event (at its head). Links which do not fail during an execution are said to be *non-faulty* during the execution. We stress that during the execution the processors may not be able to detect whether a link is faulty or not. The statement that a link is faulty is transcendental to the network (i.e., it is made once the execution terminates and by an outside observer). We make no restrictions on the number of faulty links during an execution. Instead we require that during every execution the directed subgraph, defined by the non-faulty links, remains strongly connected.

Our lower bound applies also to the restricted case in which links are *fail-stop*, namely for each link the sequence of send-message

events having corresponding receive-message events is a prefix of the sequence of send-message events. Furthermore, it applies also to networks with bi-directional links vulnerable to uni-directional faults (such networks can be represented as directed graphs with anti-parallel edges).

Our complexity measure for computational tasks is the number of messages sent during the "worst" execution of the "best" protocol achieving the task. Namely, we consider all protocols achieving the task, and for each protocol we consider all possible local inputs and all possible executions of the protocol on these inputs.

The computational task we consider is called Global Computation. *Global Computation* of a function f is the task terminating so that one designated processor outputs $f(v_1,...,v_n)$, where v_i is the local input of processor i, and n is the number of processors in the network. The task is initiated by the same designated processor. It can be easily shown that other versions of the task, such as the version in which the result of computation has to be obtained in all processors, are not easier (and are not much harder either).

The complexity of global computation of a function f may depend on the function f itself. However, this dependency is not the focus of the current paper. In particular, we are not interested in "degenerate" functions which do not depend on all their inputs. Instead, we are interested in the complexity of global computations of *input sensitive* functions. An n-ary function f is called *input sensitive* if there exists a sequence of values $(v_1,...,v_n)$ such that for every i there exists a u_i such that

$$f(v_1,...,v_{i-1},v_i,v_{i+1},...,v_n) \neq f(v_1,...,v_{i-1},u_i,v_{i+1},...,v_n)$$

For example, SUM is input sensitive, and so are AND, MAX, and many other natural functions.

3. A LOWER BOUND FOR THE COMPLETE GRAPH

In this section we present a tight lower bound, on the complexity of Global Computation, for the case of a complete communication graph. This communication graph consists directed edges between every two vertices (in both directions). We show that in an execution of any algorithm, which performs Global Computation on a complete graph with $4n+1$ vertices, at least n^3 messages are sent.

Throughout the rest of this section we denote by $G(V,E)$ the complete graph, and assume that $|V|=4n+1$, for some n. We denote the predetermined processor which initiates the execution by I and partition the rest of the $4n$ vertices into three subsets. The vertices of the first subset are called *starters* and are denoted by $\{S_i\}_{i=1}^{n}$. The vertices of the second subset are called *transmitters* and are denoted by $\{T_i\}_{i=1}^{2n}$. The vertices of the last subset are called *receivers* and are denoted by $\{R_i\}_{i=1}^{n}$ (the names given to the vertices are derived from their role in the following proof).

From now on we are interested only in a subset of the graph's edges, so we drop all the other edges from the graph. One may assume that all these edges are faulty in every execution, and that no message sent over such edge reaches its destination. The edges which remain are (see figure 3.1) as follows: For every starter S_i, we keep the edge (I,S_i). We call these edges *trigger* edges (the meaning of an edge's name is related to its role in the proof). For every starter S_i and every transmitter T_j, we keep the edge (S_i,T_j). We call these edges *routing* edges. For every transmitter T_i and every receiver R_j, we keep the edge (T_i,R_j). We call these edges *charge* edges. We also keep for every vertex in the graph, $v \in V-\{I\}$, the edges (I,v) and (v,I). These edges are called *auxiliary* edges and are not shown in Figure 3.1. Note that the set of auxiliary edges contains the set of trigger edges.

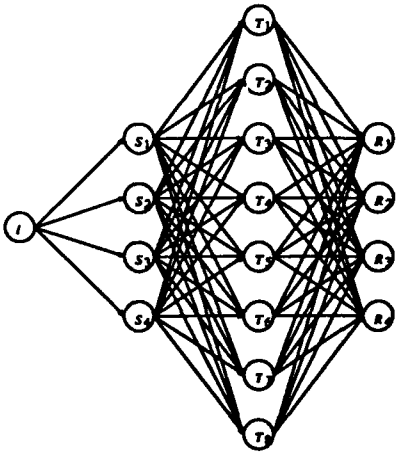


Figure 3.1: the graph's structure for $n=4$

Any algorithm which performs Global Computation on the graph G, must work correctly for any scheduling of the messages delays (as long as it agrees with the FIFO rule for each edge). In our proof we present a scheduler which causes sending at least n^3 messages during an

execution of any such algorithm.

From now on we fix an algorithm A (initiated by the processor I) which performs Global Computation on the graph G and describe the scheduler's strategy. At the beginning of the execution, the scheduler delays all messages sent over the trigger edges and enables arrival of messages sent over all other edges (in an arbitrary order) until a *stable* state of the network is reached.

Definition 3.1: The execution reaches a *stable* state when there are no more messages in transmit over the non-delayed edges (i.e., every message sent over a non-delayed edge is received at its other end).

When a stable state of the network is reached the scheduler "releases" one of the delayed edges (i.e., this edge is no longer delayed and arrival of messages sent over it is enabled). The scheduler waits till the execution reaches a stable state again. We stress that an edge which had been released is never delayed again during this execution. The scheduler continues this process (of releasing delayed edges) until all the n trigger edges are released. Thus, we have n such phases, from the first release of a delayed trigger edge, till the execution of algorithm A is ended (Global Computation can not be completed before all trigger edges are released, because the initiator I needs to get information from all the starters in order to compute the desired input-sensitive function and the starters do not send messages before receiving a message over their incoming trigger edges).

From now on we confine ourselves to schedulers as described above and only take advantage on our freedom to choose the order in which the delayed edges are released. Although this is a special case of scheduling, the messages complexity of algorithm A is measured over all possible schedules and therefore proving lower bound, for these schedules only, suffices.

Claim 3.1: Suppose that the execution (of the algorithm) is in a stable state and that the set of delayed edges is not empty. Then, there exists a delayed edge and a corresponding schedule (which starts by the release of this edge) such that this schedule yields sending at least n^2 messages (over the charge edges) before the next stable state is reached.

In order to prove Claim 3.1, we first prove Claim 3.2 below.

Definition 3.2: Suppose that the execution is in a stable state and that (I,S_i) is a trigger edge which is delayed at this state. A routing edge (S_i,T_j), for some $1 \leq j \leq 2n$, is called *good*, if releasing the trigger (I,S_i) and scheduling message delivery only along the path $I \rightarrow S_i \rightarrow T_j$ causes T_j to send messages over all its n outgoing charge edges. Otherwise the routing edge (S_i,T_j) is called *bad*.

We stress that an edge is defined good with respect to a specific stable state. Also, in defining the edge (S_i,T_j) as either good or bad, we consider only the behavior of T_j in schedules in which (after the current stable state) both S_i and T_j receive messages only from I and S_i respectively. Finally, if (S_i,T_j) is bad it means that there exists a receiver R_k so that if we schedule events only along the path $I \rightarrow S_i \rightarrow T_j \rightarrow R_k$ no message from S_i will reach R_k (i.e., T_j will not send a message to R_k). In this case, the edge (T_j,R_k) is called a *blocking* edge for the edge (S_i,T_j).

Claim 3.2: Suppose that the execution (of the algorithm) is in a stable state and that the set of delayed edges is not empty. Then, there exists a delayed edge (I,S_i) such that S_i has at least n good outgoing routing edges (out of its $2n$ outgoing routing edges).

Proof of Claim 3.2: Let us assume to the contrary that the execution is at a stable state and for every delayed edge (I,S_i), the vertex S_i has more than n outgoing routing edges which are bad. We can assign each delayed edge one of its corresponding bad routing edges so that these bad edges form a matching (i.e., each S_i is assigned to a different T_j). Such an assignment exists because there are at most n delayed edges and every delayed edge has more than n possible corresponding bad routing edges. Let M denote the above resulting matching.

With respect to the above stable state, we describe a schedule that contradicts the correctness of algorithm A. The schedule consists of a setting of the edges to either fail-stop (at this stage) or non-faulty. Following is a description of this setting.

Let (S_i,T_j) be in the matching M and let (T_j,R_k) be a blocking edge for (S_i,T_j) (a blocking edge must exist because (S_i,T_j) is bad). Then we set the edges (S_i,T_j) and (T_j,R_k) to be

non-faulty and set all other outgoing edges of of S_i and T_j to be fail-stop. We do the same for every routing edge of the matching M (note that the R_k's chosen by this procedure do not have to be distinct). In addition, all (incoming and outgoing) edges of starters or transmitters, not participating in the matching, are set to be non-faulty. Finally, all (incoming and outgoing) edges of the receivers are non-faulty.

We first show that the graph composed of the non-faulty edges is strongly connected. We show for each vertex in G a directed circuit, composed of non-faulty edges, containing the initiator I. Each vertex which participates in one of the paths $I \rightarrow S_i \rightarrow T_j \rightarrow R_k$ mentioned above, participates in the directed circuit $I \rightarrow S_i \rightarrow T_j \rightarrow R_k \rightarrow I$. Any other vertex $v \in V - \{I\}$ which does not participate in any of these circuits, participates in the directed circuit $I \rightarrow v \rightarrow I$ which consists of its auxiliary edges.

We now show that in this setting, Global Computation can not be performed. Suppose we release now all the delayed edges. Clearly, messages sent over the delayed edge (I,S_i) and arriving at S_i can only cause delivery of messages over (S_i,T_j), since this is the only non-faulty outgoing edge of S_i. The transmitter T_j (which (S_i,T_j) is its only non-faulty incoming edge) would not send a message over the blocking edge (T_j,R_k) which is its only non-faulty outgoing edge. Therefore, no message is sent from a vertex, participates in the matching M, to any vertex outside the matching. Since this process started from a stable state no other vertex in the network can send messages. Clearly, Global Computation is not completed, because the initiator received no information through this process. Thus, we have reached a contradiction. □

Proof of Claim 3.1: By Claim 3.2 we get that there is a delayed edge (I,S_i) such that S_i has at least n good outgoing routing edges (out of its $2n$ outgoing routing edges). When releasing the delayed edge (I,S_i), the vertex S_i (among other things) sends messages over its n good outgoing routing edges before getting any other messages (we postpone delivery of other messages because (delivering) otherwise we are not guaranteed that S_i would send these messages). Each transmitter at the end of such good edges sends messages over all its n outgoing charge edges (as guaranteed by the definition of a good edge). Therefore, we have at least n^2 messages sent over charge edges before the next stable state is reached. □

By Claim 3.1 we get that the release of an appropriate edge (by the scheduler), each time the execution reaches a stable state, causes sending at least n^2 messages over charge edges before reaching a stable state again. Since we have n trigger edges, during the whole execution of algorithm A, at least n^3 messages are sent over charge edges.

Theorem 1: Let Π be an arbitrary protocol, for computing the sum (or any other input-sensitive function) of the local values residing in the processors of a complete network with n processors and uni-directional links (which may fail-stop in an undetectable manner). Then there exists an execution of Π, on the network, in which $\Omega(n^3)$ messages are sent.

4. A LOWER BOUND FOR THE GENERAL CASE

The result of the previous section can be restated as follows: The message complexity of Global Computation on dense graphs with n vertices and $m = O(n^2)$ edges, is $\Omega(n \cdot m)$. Our aim is to generalize this result to graphs which are sparser. Namely, our aim is to prove that for every n and $m = m(n)$ the message complexity of Global Computation is $\Omega(n \cdot m)$. Actually, we only obtain a $\Omega\left(\frac{n \cdot m}{L(n)}\right)$ bound where L is a polylogarithmic function. To this end we modify the graph used in the previous section as follows. Instead of having $O(n^2)$ charge edges we have only $O(m)$ such edges which connect the $O(n)$ transmitters with the $\frac{m}{n}$ receivers. In addition, we need to decrease the number of routing edges. Loosely speaking, this is done by replacing the $O(n^2)$ routing edges by a sparse routing gadget, which is based on the sparse routing graph presenting below.

4.1 The sparse routing graph

A *sparse routing graph* of size n is an acyclic directed graph which has n vertices with indegree zero called *sources*, and $2n$ vertices with outdegree zero called *targets*. All other vertices in the graph are called *intermediate* vertices. In addition the graph should satisfy the following routing properties:

(1) From every source of the graph there exists a unique directed path to each target. Furthermore, the paths from each source, to all targets, form a directed tree (in which the source vertex is its root).

(2) The graph contains at most $2n \cdot \log_2(4n)$ vertices and at most $4n \cdot \log_2(4n)$ edges.

(3) Suppose that each source randomly selects (uniformly and independently) one of the $2n$ possible targets. Then the probability that one of the induced source-target paths crosses more than $2\log_2^2(n)$ of the other paths is bounded above by $1/2$ (In case two paths share i of their vertices we say that they cross each other i times).

Sparse routing graphs do exists (see Appendix). We remark that the choice of constants in the above definition is not essential to the lower bound proven below.

4.2 The graph for which the lower bound is proven

Let $k \triangleq \log_2(n)$. We now describe the sparse gadget (based on the sparse routing graph). We first replace every non-source in the sparse routing graph by $D = 5\log_2^2(n) = 5k^2$ vertices. Then we replace each directed edge of the sparse routing graph by the set of directed edges between all pairs of vertices corresponding to the original edge. This completes the construction of one *layer*. Note that the layer contains many directed paths which correspond to a single source-target path in the original sparse routing graph. Any such path is called a *routing* path and the set of these paths is called a *super-path*. Our gadget is composed of k layers which share their sources (i.e., the gadget has n sources attached to k distinct layers).

Using this gadget we can present the communication graph $G(V,E)$ which we use in our proof. Again we denote the predetermined initiator by I. The n sources of the gadget are the *starters* denoted by $\{S_i\}_{i=1}^{n}$. The $2k \cdot D \cdot n$ targets of the gadget are the *transmitters* denoted by $\{T_i\}_{i=1}^{2kDn}$. We also have in G the set of $\frac{m}{n}$ *receivers* denoted by $\{R_i\}_{i=1}^{m/n}$. The graph also contain the internal vertices of the sparse routing gadget. The edges of G (in addition to the internal edges of the gadget) are as before: For every starter S_i, we have the edge (I,S_i). We call these edges *trigger* edges. For every transmitter T_i and every receiver R_j, we have the edge (T_i,R_j). We call these edges *charge* edges. Again we also have for every vertex in the graph, $v \in V - \{I\}$, the edges (I,v) and (v,I). These edges are called *auxiliary* edges (we stress that the internal vertices of the gadget also have auxiliary edges). Note that in the graph G we have routing paths, from starters to transmitters, instead of the routing edges of the graph of Section 3.

Since the sparse routing graph contains at most $2n\log_2(4n)$ vertices and at most $4n\log_2(4n)$ edges, we get that the graph G contains $O(n\log_2^4(n))$ vertices and $O(m\log_2^3(n)) + O(n\log_2^5(n))$ edges.

4.3 Lower bound argument

Again, we fix an algorithm A which perform Global Computation on the graph G and "play" the role of the scheduler. The scheduler's behavior here is very similar to its behavior in Section 3. The set of the *delayed edges* contains again, at the beginning of the execution of the protocol, the n trigger edges. This time we show that each release of a delayed edge by the scheduler, in a stable state of the network, causes sending at least $1/4m$ messages over charge edges. This yields a $\Omega(n \cdot m)$ lower bound as required.

Claim 4.1: Suppose that the execution (of the algorithm) is in a stable state and that the set of delayed edges is not empty. Then, there exists a delayed edge and a corresponding schedule (which starts by the release of this edge) such that this schedule yields sending at least $1/4m$ messages (over the charge edges) before the next stable state is reached.

The proof of this claim follows from Claim 4.2.

Definition 4.1: Suppose that the execution is in a stable state and that (I,S_i) is a trigger edge which is delayed at this state. A routing path, which starts at S_i and ends at T_j, for some j, is called *good*, if releasing the trigger (I,S_i) and scheduling message delivery only along the path $I \rightarrow S_i \rightarrow \cdots \rightarrow T_j$, causes T_j to send messages over all its $\frac{m}{n}$ outgoing charge edges. Otherwise the routing path $S_i \rightarrow \cdots \rightarrow T_j$ is called *bad*.

Note that if the trigger edge (I,S_i) is delayed, the state of the network is stable and the routing path $S_i \rightarrow \cdots \rightarrow T_j$ is bad, then there exists a charge edge (T_j,R_k) such that when releasing the edge (I,S_i) (and scheduling message delivery only along the path $I \rightarrow S_i \rightarrow \cdots \rightarrow T_j$)

the vertex, T_j does not send any message over the charge edge (T_j,R_k). We call this charge edge a *blocking* edge of the routing path $S_i \rightarrow \cdots \rightarrow T_j$.

Definition 4.2: A super-path is called *good* at the current stable state if at least half of the routing paths corresponding to it are good (at the current stable state). Otherwise, the super-path is called *bad* at the current stable state.

Note that all the routing paths corresponding to a specific super path belong to the same layer of the routing gadget.

Claim 4.2: Suppose that the execution (of the algorithm) is in a stable state and that the set of delayed edges is not empty. Then, there exists a delayed edge (I,S_i) such that S_i has at least $\frac{1}{2}n$ good outgoing super-paths (out of its $2nk$ outgoing super-paths).

Proof of Claim 4.2: Let us assume to the contrary that the network is at a stable state and for every delayed edge, (I,S_i), there are more than $\frac{3}{2}n$ super-paths in every layer of the super graph which are bad.

Consider the following process on the first layer of the routing gadget: Each starter, S_i, randomly selects one of its outgoing super-paths out of the $2n$ possible super-paths at the first layer. Since there are more than $\frac{3}{2}n$ bad outgoing super-paths for every starter in this layer, we get that the probability that the starter selects a bad super-path is greater than $\frac{3}{4}$. Hence, using Markov's inequality, we get that with probability grater than $\frac{1}{2}$, at least half of the starters select bad super-paths. By the properties of the super-paths of the layer we get that the probability that one of the selected super-paths crosses more than $2\log_2^2(n)$ other selected super-paths is bounded by $\frac{1}{2}$. Combining the above two facts we get that there exists a possible choice in which at least half of the starters select bad super-paths and none of these selected super-paths crosses more than $2\log_2^2(n)$ other selected super-paths.

By this process we have selected bad super-paths for half of the starters. Applying the same process for the remain starters on the second layer of the gadget yields bad super-paths for half of them. Thus, By using all $k=\log_2(n)$ layers of the gadget we can select for each starter a bad super-path in a way that none of these selected super-paths crosses more than $2\log_2^2(n)$ other selected super-paths.

Our next step is to choose for each starter one of the bad routing paths, corresponding to its bad super-path, in a way that all these paths are vertex disjoint. This process can be done sequentially (i.e., for starter after starter). Suppose we try to choose a routing path, for a new starter, out of the paths corresponding to its super-path. In order to choose a vertex disjoint routing path for the new starter, we have to avoid paths containing vertices which participates in routing paths chosen for previous stages. We use the fact that each super-path crosses at most $2\log_2^2(n)$ of the other super-paths. Each such crossing rules out at most a $\frac{1}{D}$ fraction of the possible paths corresponding to the current super-path. Hence, at most a $2\log_2^2(n)\cdot\frac{1}{D}=\frac{2}{5}$ fraction of the paths are ruled out. Since, at least half of the routing paths of the super-path are bad, we can choose a bad routing path which does not cross any of the previous ones.

With respect to the above stable state, we again describe a schedule contradicts the correctness of algorithm A. The schedule consists of a setting of the edges to either fail-stop (at this stage) or non-faulty.

Let $I \rightarrow S_i \rightarrow \cdots \rightarrow T_j$ be a path formed by the delayed edge (I,S_i) and the bad routing path chosen for S_i. Let (T_j,R_k) be a blocking edge for that bad routing path. Then we set the edges of the bad routing path, along with the edge (T_j,R_k), to be non-faulty and set all other outgoing edges of the vertices of the bad routing path to be fail-stop. We do the same for every chosen bad routing path. Again, all edges (incoming and outgoing) between vertices not on any of these paths are set to be non-faulty.

We first show that the subgraph consisting only the non-faulty edges is strongly connected. We show again for each vertex in G a directed circuit, composed of non-faulty edges, containing the initiator I. Each vertex which participates in one of the paths $I \rightarrow S_i \rightarrow \cdots \rightarrow T_j \rightarrow R_k$ where $S_i \rightarrow \cdots \rightarrow T_j$ is the bad routing path chosen for S_i, participates in the directed circuit $I \rightarrow S_i \rightarrow \cdots \rightarrow T_j \rightarrow R_k \rightarrow I$. Any other vertex $v \in V-\{I\}$ which does not participate in any of these circuits, participates in the directed circuit $I \rightarrow v \rightarrow I$ which consists of its auxiliary edges.

We now show that in this setting, Global Computation can not be performed. Suppose we release now all the delayed edges. Clearly, messages sent over the delayed edge (I,S_i) and

arriving at S_i can only cause delivery of messages over the (chosen) path $S_i \rightarrow \cdots \rightarrow T_j$, since it composed of the only non-faulty edges reachable from S_i. The transmitter T_j (which the last edge of this path is its only non-faulty incoming edge) would not send message over the blocking edge (T_j,R_k) which is its only non-faulty outgoing edge. Therefore, no message is sent from a vertex, participates in one of the paths constructed above, to any vertex outside these paths. Since this process started from a stable state no other vertex in the network can send messages. Clearly, Global Computation is not completed, because the initiator received no information through this process. Thus, we have reached a contradiction. □

Proof of Claim 4.1: By Claim 4.2 we get that there is a delayed edge (I,S_i) such that S_i has at least $1/2n$ good outgoing super-paths. Consider randomly choosing for these super-paths corresponding routing paths in a way which forms a directed tree. In other words, consider the directed "super-tree" corresponding to the set of good super-paths and select uniformly a tree corresponding to this super-tree. Every routing path in that tree is good with probability at least $1/2$. Therefore, it is possible to construct such tree in which at least half of the paths are good. The schedule corresponds to the delayed edge (I,S_i) is described below. When releasing the delayed edge (I,S_i), vertex S_i sends messages over its $1/2 \cdot \frac{n}{2} = 1/4n$ outgoing good routing paths (which participates in the directed tree). Each transmitter at the end of such good routing path sends messages over all its $\frac{m}{n}$ outgoing charge edges (as guaranteed by the definition of good routing path). Therefore, at least $1/4m$ messages are sent over charge edges during this schedule (before reaching the next stable state). □

By Claim 4.1 we get that the release of an appropriate edge (by the scheduler) each time the execution reaches a stable state, causes sending at least $1/4m$ messages over charge edges before reaching a stable state again. Since we have n trigger edges, during the whole execution of algorithm A, at least $1/4n \cdot m$ messages are sent over charge edges.

Theorem 2: Let Π be an arbitrary protocol, for computing the sum (or any other input-sensitive function) of the local values residing in the processors of some network with n processors and m uni-directional links (which may fail-stop in an undetectable manner). Then there exists an execution of Π, on such a network, in which $\Omega\left(\frac{n \cdot m}{\log_2^7(n)}\right)$ messages are sent.

We get the constant 7 in the above bound, since the graph G contains $O(n\log_2^4(n))$ vertices and $O(m\log_2^3(n)) + O(n\log_2^6(n))$ edges. For $m \geq n \cdot \log_2^3(n)$ we get an $\Omega\left(\frac{n \cdot m}{\log_2^7(n)}\right)$ bound by a mere substitution. For $m < n \cdot \log_2^3(n)$ (yet $m > (1+\varepsilon) \cdot n$ for some constant $\varepsilon > 0$) a slightly more careful argument yields the stated bound. We believe that the constants (in the exponent of the logarithmic factors) can be improved significantly.

REFERENCES

[GS] Goldreich, O., and Shrira, L., "On the Complexity of Global Computation in the Presence of Link Failures: the Case of Ring Configuration", Distributed Computing, 1991.

[IR] Itai, A., and Rodeh, M., "The Multi-Tree Approach to Reliability in Distributed Networks", *Proc. of the 25th IEEE Symp. on Foundation Of Computer Science*, 1984, pp. 137-147.

[SR] Shrira, L., and Rodeh, M., "Methodological Construction of Reliable Distributed Algorithms", TR-361, Comp. Sc. Dept., Technion Israel, 1985.

[U] Upfal, E., "Efficient Schemes for Parallel Communication", *Symposium on Principles of Distributed Computing* 1982, pp. 55-59

APPENDIX

We present a graph $G_k(V_k,E_k)$, which is a sparse routing graph of size $n=2^k$ (for some k). The graph presented here is based on the balanced communication scheme presented in [U]. Fig. A.1 shows the graph G_k for $n=8$.

The graph G_k is composed of $k+2$ layers of $2n$ vertices each. The vertices at every layer are labeled with binary strings of length $k+1$. Edges in G_k exist only between every two adjacent layers. In particular, the following edges

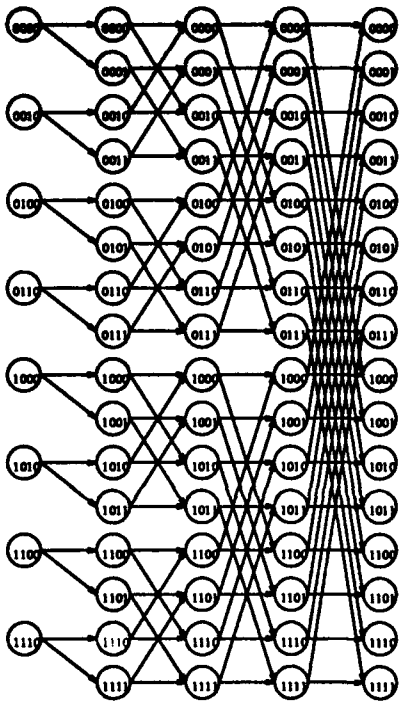


Fig. A.1 The graph G_k for $n=8$

exist between layer i and layer $i+1$ (where $1 \le i \le k+1$):

- From every vertex in layer i to the vertex with the same label in layer $i+1$.
- From every vertex in layer i to the vertex (in layer $i+1$) labeled with the same binary string except for the i-th bit which is toggled.

Note that in the above graph there are $2n$ source vertices (in layer 1) and $2n$ target vertices (in layer $k+2$). Therefore, in order to comply with the definition of routing graph, we drop, from G_k, all the source vertices in which the first bit of their label is 1 (along with their outgoing edges).

Note that from every source of the sparse routing graph there exists a unique directed path to each target and that the paths from each source, to all targets, form a directed tree.

We denote the process of each source randomly selects (uniformly and independently) one of the $2n$ possible targets by a *random selection.*

Claim A.1: In a random selection on the graph $G_k(V_k,E_k)$, for each target in V_k the probability to be chosen c (or more) times is bounded by 2^{-c}.

Proof of Claim A.1: An exact calculation of this probability gives:

$$\sum_{i=c}^{n}\binom{n}{i}\cdot\left(\frac{1}{2n}\right)^{i}\cdot\left(1-\frac{1}{2n}\right)^{n-i}$$

Performing an indices transformation $j \leftarrow i-c$ we get:

$$\sum_{j=0}^{n-c}\binom{n}{j+c}\cdot\left(\frac{1}{2n}\right)^{j+c}\cdot\left(1-\frac{1}{2n}\right)^{n-c-j}$$

Using the combinatorial inequality:

$$\binom{n}{j+c} < \binom{n-c}{j}\cdot n^{c}$$

we can bound our probability by:

$$n^{c}\cdot\left(\frac{1}{2n}\right)^{c}\cdot\sum_{j=0}^{n-c}\binom{n-c}{j}\cdot\left(\frac{1}{2n}\right)^{j}\cdot\left(1-\frac{1}{2n}\right)^{n-c-j}$$

which is smaller than 2^{-c}. □

Claim A.2: In a random selection on the graph $G_k(V_k,E_k)$, for every vertex (not necessarily a target vertex) the probability of participating in c (or more) of the induced source-target paths is bounded by 2^{-c}.

Proof of Claim A.2: Consider a random selection on the graph G_k. By dropping all the vertices in the last i layers along with their incoming edges, we get 2^i separated subgraphs (each identical to G_{k-i}). The original random selection on the graph G_k, induces random selections on each one of these subgraphs. By Claim A.1, we get that for each one of the new targets, the probability to be chosen c (or more) times is bounded by 2^{-c}. Therefore, in the original random selection, the probability of its participating in c (or more) of the induced source-target paths is also bounded by 2^{-c}. □

By noting that the graph G_k contains $2n\log_2(2n)$ vertices which are not source vertices and that the length of each unique path is bounded by $k+1$, we can derive From Claim A.2 the following:

Corollary A.1: In a random selection on the graph G_k, the probability that there exists a vertex in the graph which participates in more then $2\log_2(n)$ of the induced selected paths is bounded by ½.

Corollary A.2: In a random selection on the graph G_k, the probability that one of the induced selected paths crosses more than $2\log_2^2(n)$ other selected paths is bounded by ½.

Note that the graph G_k contains less than $2n\log_2(4n)$ vertices and less than $4n\log_2(4n)$ edges and therefore G_k is a sparse routing graph.

Observing Self-Stabilization

Chengdian Lin
University of Chicago
lin@cs.uchicago.edu

Janos Simon
University of Chicago
simon@cs.uchicago.edu

1 Abstract

Self-stabilizing systems have been proposed as a desirable method of achieving fault tolerance. They are guaranteed to eventually eliminate any initial set of errors. This also implies that *infrequent* errors can be dealt with. For more details see [1-7]. In this paper we study deterministic self-stabilizing algorithms for leader election in rings. We have two sets of contributions. First, we introduce the formal definition of an *observer* at each location: a local process that can detect correctness, but cannot influence the protocol. Every self-stabilizing algorithm can have such associated observers. We believe that this is a good abstraction. We also claim that some such notion is necessary to make self-stabilizing protocols useful.

The notion of an observer suggests a natural question to anyone familiar with the P vs. NP question: are there situations in which it is easier to *detect* stability than to *achieve* it?

We exhibit a somewhat contrived problem where this is indeed the case. Our second contribution is a careful study of deterministic leader election algorithms on uniform rings of processors. We show that the class of protocols based on the general ideas of Burns and Pachl [2] require $\Theta(n^3)$ steps in the worst case to become stable. We present a protocol for this problem that detects stability in $\Theta(n^2)$ steps, and uses only five extra bits. Thus, for this class of algorithms verification is easier than computation.

We also characterize exactly the memory requirements of these algorithms. The algorithm in [2] requires $\Omega(p^2/\ln p)$ states per processor for a ring of size p. We give a combinatorial characterization of the exact number of states needed for similar protocols as $O(pR(p))$, where $R(n) \leq O(\sqrt{n/\ln n \ln\ln n})$ is a Ramsey theoretic function.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0113...$1.50

2 Introduction

Self-stabilizing protocols are an elegant methodology to achieve fault tolerance. While there has been a large number of recent papers on the topic, there are several problems with the technique. Self-stabilization requires that the system enter the desired stable set of configurations, no matter what the initial state of individual processors (and no matter in which order enabled processors take steps). It follows that although stability is eventually attained, a processor cannot determine whether this has yet occurred (since part of the incorrect initial configuration could be the setting of the components of the state of the processor that indicate whether the system is stable.)

As a concrete example, consider a token ring. Once stability has been attained, there is a single token in the ring, and it can be used for granting control to the processor that has it. Before the ring is stable, there may be several tokens that are used by the the self-stabilizing protocol. During this period, a processor that has a token may not assume that it is the unique enabled process. Since processors cannot know whether the ring is stable, all actions must be tentative, even when the ring is already in a legitimate configuration.

We propose a new model, in which it is meaningful to say that "a processor knows that the ring is stable". Formally, this statement will mean the following:

> the state set of each processor is of the form $State_{SSP} \times Observe$ (SSP stands for self-stabilizing protocol). There is a subset

Stable of *Observe* such that for a correct protocol, if the ring stabilizes, eventually every processor will be in a state in the set $States_{SP} \times Stable$. Conversely, if a processor is in state belonging to this set, the ring is stable (provided no new errors have occured).

In order for the definition to make sense, we stipulate that

- There is a value $Init \in Observe$ such that all processors start in a state in $States_{SP} \times Init$. *The daemon cannot change the Observe component of the state.*
- The transition relation *does not depend on the Observe component.* It is solely a function of $States_{SP}$.

This formalizes the notion that there is a component of the processor state that is used solely for observation. It cannot influence the protocol, but it is not affected by the daemon – so it can always have a meaning.

We shall see that this can be implemented (in the case of uniform rings) by very few extra bits, and thus it might be of practical use. One might think of scheme as an outside observer that looks at the processor and determines when the ring is stable. There may be a delay between the time the ring becomes actually stable, and the time the observer detects it. Since the protocol cannot be influenced by the observer, the observation routine runs independently from the protocol: thus the assumption that the observer starts in a ('correct') unique initial state is not very restrictive.

Of course, the observer indicating the ring to be stable means only that at some time in the past the ring was stable. Provided that no new errors occurred, the ring is currently stable.

There is a wealth of interesting questions about this model. For example

- How easily can stability be detected?
- How many extra states are needed?
- Can this be made useful?

In the rest of this paper we deal with deterministic leader election protocols in uniform (anonymous) rings. As Dijkstra [4] pointed out, there is no self-stabilizing solution for n identical processors in a ring, if n is composite. However, if n is prime, Burns and Pachl [2] exhibited self-stabilizing election protocols, with as little as $O(n^2/\ln n)$ states per processor.

We will call protocols based on the elegant ideas of [2] *BP type protocols.* We show that a BP type protocol is guaranteed to attain stability after $4n^3$ moves, and that there is an adversary strategy for the daemon that makes the protocol run $\frac{1}{2}(n-1)^3$ steps before stabilization.

We then study stability detection in this model. There is an obvious simple observer strategy: use a counter that is initially 0, and is incremented at every move. When its value reaches an appropriate number (of $O(n^3)$) we know that the ring is stable. This strategy has many faults: it uses too many states, and it is nonadaptive – it uses a large amount of time even for a ring that is initially stable. What our bounds show is that after $O(n^3)$ steps the ring becomes stable. Since the ring is asynchronous it is not even clear in principle how many steps need to be taken before every processor takes $O(n^3)$ steps (in fact $O(n^3)$ suffice.) Note that it is unclear how to get a global count - the counters belong to the observer, so they cannot participate in the protocol. Thus, we cannot transmit individual counters (unless we were willing to use ones that can be altered by the adversary - in which case it is unclear how we could keep them correct.)

We show that there is a much better observer strategy by presenting a protocol which correctly detects the stability of ring after $O(n^2)$ moves from the time the ring is stabilized. The observer uses only 5 bits. Finally, we show that the $O(n^2)$ detection delay is necessary.

These two results show that in this case checking stability is easier than achieving it.

Finally, we characterize the number of states per processor that are necessary for a correct implementation of a BP type protocol. Let $R(n)$ be an integer with the following property: it is possible to color all integers in $D = \{1, 2, \cdots, n-1\}$ with $R(n)$ colors, such that for any subset $\{a_1, a_2, ..., a_k\}$ of D, $1 < k < n$, if all a_i have same color then $\sum_{i=1}^{k} a_i \neq n$. There is a BP type protocol with $nR(n)$ states. In particular, the optimum protocol has as many states as the smallest possible number of colors. Determining this number is an interesting open problem of combinatorics. We have no nontrivial lower bounds. We are able to produce a coloring with $O(\sqrt{\frac{n}{\ln n \ln\ln n}})$ colors. This yields $O(n\sqrt{\frac{n}{\ln n \ln\ln n}})$ states, as opposed to $O(n^2/\ln n)$ in [2].

3 BP type Deterministic Self-Stabilizing Protocols for Asynchronous Rings of Prime Size

We use the formal definitions of Burns and Pachl. For simplicity, we deal with unidirectional rings. For the more general definition, refer to [2] .

An *n-processor ring* is a four-tuple, $S = (G, R, \Gamma, \Delta)$ where G is a cycle, the processors are denoted by $0, 1, \cdots, n-1$ in order around the cycle, and R orders the processors so that the first (leftward) neighbor of processor i is processor $i-1 \pmod n$, and the second (rightward) neighbor is processor $i+1 \pmod n$. $\Gamma = \Sigma_0 \times \cdots \times \Sigma_{n-1}$ is a set of configurations with finite sets $\Sigma_0, \cdots, \Sigma_{n-1}$,$\Delta = \delta_0, \cdots, \delta_{n-1}$ is a sequence of transition relations. We refer to Σ_i and δ_i as the state set and transition relation, respectively, *of processor i*, or P_i. Each δ_i is a relation from $\Sigma_{i-1} \times \Sigma_i$ to Σ_i. Thus the ring is unidirectional.

Let $\gamma = (a_0, a_1, \cdots, a_{n-1}) \in \Gamma$ be a configuration. We denote by $\delta_i(a_{i-1}, a_i)$ the set $\{x | (a_{i-1}, a_i, x) \in \delta_i\}$. Then we define $\delta_i(\gamma)$ to be

$$\{a_0, a_1, \cdots, a_{i-1}, x, a_{i+1}, \cdots, a_{n-1}) \mid x \in \delta(a_{i-1}, a_i)\}.$$

Processor i is *enabled* at γ if $\delta_i(\gamma)$ is not empty. If processor i is enabled at γ, and $\gamma' \in \delta_i(\gamma)$, we write $\gamma \xrightarrow{i} \gamma'$ and say that $\gamma \xrightarrow{i} \gamma'$ is a step of P_i. The notation $\gamma \rightarrow \gamma'$ means that $\gamma \xrightarrow{i} \gamma'$ for some i. A computation of a system is a finite or infinite sequence $\gamma_0\gamma_1\cdots$ such that $\gamma_{j-1} \rightarrow \gamma_j$ for all j. Thus, we consider only the serial computations. Only one processor takes a step at a time, and each step in the computation depends on the configuration resulting from the previous step in the sequence. If more than one processor is enabled at a configuration γ, then the central demon (scheduler) will select one of the enabled processors to take a step.

Definition 1. A ring $S = (G, R, \Gamma, \Delta)$ is *self-stabilizing* if and only if there is a set $\Lambda \subseteq \Gamma$, called the *legitimate configurations* of S, such that the following conditions are satisfied:

1. [No Deadlock] For every $\gamma \in \Gamma$ ther is a $\gamma' \in \Gamma$ such that $\gamma \rightarrow \gamma'$.
2. [Closure] For every $\lambda \in \Lambda$, every λ' such that $\lambda \rightarrow \lambda'$ is in Λ.
3. [No Livelock] Every infinite computation of S contains a configuration in Λ.
4. [Mutual Exclusion] For every $\lambda \in \Lambda$, exactly one processor is enabled.
5. [Fairness] For every processor i, every infinite computation consisting of configurations in Λ contains an infinite number of steps by P_i.

Because the ring is unidirectional, we write $a\underline{b} \rightarrow d$ to express that a processor in state b (indicated by underlining) is enabled to step to state d when it has a processor in state a on the left. A protocol is specified by such rules and by conditions under which they can be applied (i.e., when the processor corresponding to state b is enabled).

A ring is uniform if $\Sigma_0 = \Sigma_1 = \cdots = \Sigma_{n-1}$, and $\delta_0 = \delta_1 = \cdots = \delta_{n-1}$. Since a solution is impossible in uniform rings of composite size, we assume n to be a prime.

Let the size of the ring be n. The states of processor are composed of two parts, the *label* and the *tag* where labels range over $L = \{0, 1, \cdots, n-2\}$ and tags over $T = \{0, 1, 2, \cdots, n\}$. We write *label.tag* to denote state with label *label* and tag *tag*.

The protocol is defined by the following two rules in which the expressions $a+1$ and $b-a$ are computed modulo $n-1$.

Rule A. If $b \neq a+1$ and ($b \neq 0$ or $t = 0$ or $t \neq f(b-a)$ or $t < u$), then

$$a.t\ \underline{b.u} \rightarrow (a+1).f(b-a)$$

Rule B. If $t \neq u$ and $a+1 \neq 0$, then

$$a.t\ \underline{(a+1).u} \rightarrow (a+1).t$$

Let Λ be the set of all cyclic permutations of configurations of the following form (underlining indicates the enabled state):

$$0.0\ 1.0 \cdots (a-1).0\ a.0\ \underline{a.0}\ (a+1).0 \cdots (n-2).0$$

for $a = 0, 1, \cdots, n-2$.

We say that the protocol is defined by function f.

Lemma 1 *If f is any function such that $f(k) = 0$ iff $k = 0$, then the protocol defined by f satisfies conditions 2,3,4, and 5 of* **Definition 1**. *[2]*

Thus, if a function f as above also satisfies condition 1 [No Deadlock], then the protocol defined by f is a correct. We shall call such functions *good*.

4 Number of Steps Used by BP Type Protocols

4.1 The Upper Bound

Theorem 1 *For any central demon and for any initial configuration, the system will enter a legitimate configuration in at most $4n^3$ steps.*

The proof of the theorem is by a careful analysis of the kinds of configurations the ring can be in. More precisely, we define the notions of gap and segment in the sequence of contiguous processors that is a configuration of the system. We then examine the action of both Rule A and Rule B on these sequences.

Definition. Let P_i and P_{i+1} be two consecutive processors with states $a.t$ and $b.u$ in a configuration γ. If $b \neq a+1 \pmod{n-1}$, then P_i and P_{i+1} form a gap of γ; the gap size (of P_i at γ) is defined to be $g(i,\gamma) = b - a \pmod{n-1}$.

Definition. A segment of γ is a maximal cyclically contiguous sequence of processors $s = (P_i, P_{i+1}, \cdots, P_{j-1}, P_j)$ which contains no gaps; the gap size of s is $g(j,\gamma)$. We will call P_i the *head* of s and P_j the *tail* of s.

A segment has the form: $a.*(a+1).*\cdots(a+k).*$. The head has label a, and the tail has label $(a+k)$.

If Rule A is applied at the head of a segment of length at least 2, then both that segment and the one to its left survive, and the size of the gap between them remains the same. Neither Rule A nor Rule B can increase the number of segments.

Note that Rule B is applied inside a segment and it does not pass a tag across a state with label 0. By changing the tail of a segment at most $n-1$ times, the tail will have state $0.f(g)$, where g is the gap size between the segment and the one on its right.

Lemma 2 *Assume there are k (>1) segments at configuration γ. Then after at most $(n+m)k$ steps of Rule A under any central demon, either there are at most $k-1$ segments left or all k segments change their tails at least m times.*

Proof. At γ, let the k segments be $s_1, s_2, \cdots, s_k$, with lengths $l_1, l_2, \cdots, l_k$ respectively. Consider segment s_k. Define the segment length array at configuration γ as

$$(l_k, l_{k-1}, \cdots, l_1)[\gamma]$$

If after p steps of Rule A, the configuration is γ_1 and the tail of segment s_k is not changed, then it is not difficult to see that:

$$p = \sum_{i=1}^{k} i(l_i - t_i)$$

where $(t_k, t_{k-1}, \cdots, t_1)[\gamma_1]$ is the segment length array at γ_1.

Assume segment s_k changed its tail at configurations $\beta_1, \beta_2, \cdots, \beta_m$ and then entered configurations $\gamma_1, \gamma_2, \cdots, \gamma_m$ respectively. Let the segment length array at β_i, $1 \leq i \leq m$ be

$$(t_{i,k}, t_{i,k-1}, \cdots, t_{i,1})[\beta_i]$$

then the segment length array at γ_i, $1 \leq i \leq m$ is

$$(t_{i,k}+1, t_{i,k-1}, \cdots, t_{i,1}-1)[\gamma_i]$$

Therefore the total number of steps of Rule A is:

$$\begin{aligned}
&k(l_k - t_{1,k}) + (k-1)(l_{k-1} - t_{1,k-1}) + \cdots + \\
&(l_1 - t_{1,1}) + 1 + \\
&k((t_{1,k}+1) - t_{2,k}) + (k-1)(t_{1,k-1} - t_{2,k-1}) + \cdots \\
&+((t_{1,1}-1) - t_{2,1}) + 1 + \\
&k((t_{2,k}+1) - t_{3,k}) + (k-1)(t_{2,k-1} - t_{3,k-1}) + \cdots \\
&+((t_{2,1}-1) - t_{3,1}) + 1 + \\
&\cdots + \\
&k((t_{m-1,k}+1) - t_{m,k}) + (k-1)(t_{m-1,k-1} - t_{m,k-1}) + \cdots \\
&+((t_{m-1,1}-1) - t_{m,1}) + 1 \\
= \;& (m-1)k + 1 + \sum_{i=1}^{k} i(l_i - t_{m,i}) \\
< \;& (m-1)k + 1 + \sum_{i=1}^{k} i l_i \\
< \;& (m-1)k + 1 + k\sum_{i=1}^{k} l_i \\
< \;& (m+n)k
\end{aligned}$$

■

A segment s of γ is *well formed* if all its tags are equal to $f(g(s,\gamma))$.

Lemma 3 *If there are k (>1) segments at configuration γ, then after at most $3kn$ steps of Rule A under any central demon, either there are at most $k-1$ segments left or all k segments are well formed.*

Proof. At γ, let the k segments be $s_1, s_2, \cdots, s_k$, with length $l_1, l_2, \cdots, l_k$ respectively. Let us consider segment s_k. Assume s_k changed its tail m times and has $0.f(g)$ (g is its gap size) as its tail at configuration β with segment length array $(t_k, t_{k-1}, \cdots, t_1)[\beta]$. By the proof of the **Lemma 2**, the protocol takes

$$(m-1)k + 1 + \sum_{i=1}^{k} i(l_i - t_i)$$

steps of Rule A. After configuration β, any new tail of segment s_k will have this same tag $f(g)$, since Rule B doesn't pass a tag across label 0. From β, after segment s_{k-1} changes its tail at most t_k times, segment s_k will have $0.f(g)$ as its head. So the segment s_k is well formed this time, say, at configuration α . By **Lemma 2**, the total steps of Rule A is then

$$\leq (m-1)k + 1 + \sum_{i=1}^{k} i(l_i - t_i) + (n + t_k)k \leq 3kn$$

■

Lemma 4 *If there are $1 < k < n$ segments at configuration γ, and all k segments are well formed, then after at most $2kn$ applications of Rule A, there will be at most $k-1$ segments left.*

Proof. At γ, let the k segments be $s_1, s_2, \cdots, s_k$, and k let the gap sizes be $g_1, g_2, \cdots, g_k$. If all $f(g_i)$ are zero, then γ is already a legitimate configuration. WLOG, assume $f(g_1) \geq f(g_2)$ and $f(g_1) \neq 0$. If the tail of segment s_1 changed m times, then the head of segment s_2 changed m times as well. But if the head of segment s_2 is $0.f(g_2)$, it can't be changed again unless the number of segments decreases. Hence, the tail of segment s_1 can be changed at most n times. By **Lemma 2**, in at most $(n+n)k$ applications of Rule A there will be at most $k-1$ segments left. ■

The following lemma provides the main tool to bound the number of applications of Rule A.

Lemma 5 *After at most $\frac{5}{2}n^3$ applications of Rule A, the system will be in a legitimate configuration.*

Proof. First application of Rule A will enter a configuration which has at most $n-1$ segments. By **Lemma 3** and **4**, the number of segments will decrease from $k(< n)$ in at most $5kn$ steps of Rule A. So the number of applications of Rule A needed before entering a one-segment configuration is at most

$$1 + \sum_{k=1}^{n-1} 5kn = \frac{5}{2}n^2(n-1)$$

In at most n more applications of Rule A, the configuration will be legitimate. ■

Now we look at Rule B.

Lemma 6 *If a segment s has length l, then Rule B can be applied at most $\frac{l(l-1)}{2}$ times before it changes its tail, and Rule B can only be applied at most l times to any new tail of s before the number of segments decreases.*

Proof. Let the segment be $a.*(a+1).*\cdots(a+l-1).*$. Then label $(a+i)$ can apply Rule B at most i times, the first part of lemma follows.

When the tail changes to $a+l$, it will have tag $f(g)$ (g is its gap size) initially. This tag can be changed at most l times. Any new tail after $a+l$ will have the same tag $f(g)$ when it joins segment s. Thus the number of times a new tail can apply Rule B is the same as the times of Rule B applied at $a+l$, which is at most l. ■

Lemma 7 *If there are $1 < k < n$ segments at configuration γ, then after at most $\frac{3}{2}n(n-1)$ applications of Rule B, there will be at most $k-1$ segments left.*

Proof. At γ, let the k segments be $s_1, s_2, \cdots, s_k$, and with length $l_1, l_2, \cdots, l_k$ respectively. For every i, Rule B can only apply to at most $n-1$ new tails of s_i, because Rule B doesn't pass a tag across label 0. Therefore, by **Lemma 6**, k will be decreased in at most

$$\sum_{i=1}^{k} \frac{l_i(l_i-1)}{2} + \sum_{i=1}^{k}(n-1)l_i < \frac{3}{2}n(n-1)$$

applications of Rule B, since $\sum_{i=1}^{k} l_i = n$. ■

The following lemma concludes the proof for Rule B.

Lemma 8 *After at most $\frac{3}{2}n^3$ applications of Rule B, the configuration will become legitimate.*

Proof. Apply **Lemma 7** at most $n-1$ times, the configuration will have only one segment. By **Lemma 6**, one-segment configuration can apply Rule B at most $\frac{n(n-1)}{2}$ times. Thus the total number of times a step of Rule B can be taken before entering a legitimate configuration is at most:

$$\frac{3}{2}n(n-1)^2 + \frac{1}{2}n(n-1) < \frac{3}{2}n^3$$

■

The two main lemmas clearly imply the **Theorem 1**.

4.2 The Lower Bound

Theorem 2 *For any correct BP type protocol P, there is a demon strategy and an initial configuration, such that P takes at least $\frac{1}{2}(n-1)^3$ steps to enter a legitimate configuration.*

The proof, is by constructing a strategy for the demon and an initial configuration which forces the protocol to run at least $\frac{1}{2}(n-1)^3$ steps before stabilization.

Let us consider the initial configuration

$$0.0\ 0.0\ \cdots\ \underline{0.0}$$

The processor with an underlined state is the processor chosen by the demon to take a move. A sequence of steps is as follows:

$$\begin{array}{ll} 0.0\ 0.0\ \cdots\ 0.0\ \underline{0.0}\ 1.0 & [\gamma_1] \\ 0.0\ 0.0\ \cdots\ \underline{0.0}\ 1.0\ 1.0 & \\ \cdots & \\ \underline{0.0}\ 1.0\ \cdots\ 1.0\ 1.0 & \\ \underline{2.f(n-2)}\ 1.0\ \cdots\ 1.0\ 1.0 & \text{by Rule B} \\ 2.0\ 1.0\ \cdots\ 1.0\ \underline{1.0} & [\gamma_2] \end{array}$$

For convenience, we rewrite configuration γ_2 as

$$1.0\ 1.0\ \cdots\ 1.0\ \underline{1.0}\ 2.0\ [\gamma_2]$$

From γ_1 to γ_2, the system takes $n-1$ steps of Rule A and one step of Rule B. Similarly, the following configurations will be reached:

$$2.0\ 2.0\ \cdots\ 2.0\ \underline{2.0}\ 3.0$$

$$\cdots$$

$$(n-3).0\ (n-3).0\ \cdots\ (n-3).0\ \underline{(n-3).0}\ (n-2).0$$

$$(n-2).0\ (n-2).0\ \cdots\ (n-2).0\ \underline{(n-2).0}\ 0.f(n-2)$$

Again, from γ_i to $\gamma_{i+1}, i = 2, 3, \cdots, n-2$, the system takes $n-1$ steps of Rule A . So far we used $(n-1)(n-2)$ steps of Rule A. After $n-1$ more steps of Rule A, the system will enter a configuration γ_n in which all segment are well formed.

$$0.0\ 0.0\ \cdots\ 0.0\ \underline{0.f(n-2)}\ 1.f(n-2)$$

In another two steps of Rule A:

$$\begin{array}{ll} 0.0\ 0.0\ \cdots\ 0.0\ 1.0\ \underline{1.f(n-2)} & \\ 0.0\ 0.0\ \cdots\ \underline{0.0}\ 1.0\ \underline{2.0} & [\beta_2] \end{array}$$

By taking $(n-2)(n-3)$ steps, the system reaches configuration

$$(n-3).0\ \cdots\ (n-3).0\ \underline{(n-3).0}\ (n-2).0\ 0.f(n-3)$$

$2(n-2)$ more steps will let system enter a strongly well formed configuration

$$0.0\ \cdots\ 0.0\ \underline{0.f(n-3)}\ 1.f(n-3)\ 2.f(n-3)$$

3 more steps, we have a configuration which has $n-4$ segments of length 1 and one segment of length 4.

$$0.0\ \cdots\ \underline{0.0}\ 1.0\ 2.0\ 3.0\ [\beta_3]$$

Let configuration $\beta_i, i = 2, 3, \cdots, n-2$, be

$$0.0\ \cdots\ \underline{0.0}\ 1.0\ 2.0\ \cdots\ i.0\ [\beta_i]$$

We know that from β_i to β_{i+1}, system takes at least $(n-i)(n-1)$ steps of Rule A. Therefore the system takes at least

$$\sum_{i=3}^{n-1}(n-i)(n-1) \geq \frac{1}{2}(n-1)^3$$

steps of Rule A to enter a legitimate configuration.

5 Stability Detection Protocol

5.1 The Protocol

If the ring is stable, the state of any fixed processor increases by one (mod n) when the processor is enabled next time. If this happens at a processor n consecutive times, its observer should be able to detect the stability. This is the idea of our detection protocol.

We present a protocol for observers to detect the stability of the ring. It uses 5 extra bits: two bits b_1, b_2 that are determined by the last move of the processor, and a three-bit counter C.

The protocol is:

The observer O initializes its counter C to 0. If its processor P takes a step of Rule A of the form $a.t\ \underline{b.u} \rightarrow (a+1).f(b-a)$, P sets

$$b_1 = \begin{cases} 1 & \text{if } b = a \\ 0 & \text{otherwise} \end{cases}$$

$$b_2 = \begin{cases} 1 & \text{if } a+1 \equiv 0 \pmod{\frac{n-1}{2}} \\ 0 & \text{otherwise} \end{cases}$$

When b_1, b_2 changes, the observer O updates its counter C according to the following rules:

$$\begin{cases} C = 0 & \text{if } b_1 = 0 \\ \text{Does Nothing} & \text{if } (b_1 = 1) \wedge ((b_2 = 0 \wedge C < 3)) \vee C = 5) \\ \text{Increases } C \text{ by 1} & \text{if } b_1 = 1 \wedge ((b_2 = 1 \wedge C < 3) \vee 3 \leq C < 5 \end{cases}$$

The observer knows that the ring is stable when $C = 5$.

5.2 The Correctness

The main ideas are the following: $C = 3$ when the ring consists of a single segment. When C becomes 4 state 0 has a tag of 0. The next time C is incremented all tags are 0, and the ring is stable. The main difficulty of the proof is showing that the counter behaves appropriately as the ring stabilizes. This is done by

proving by induction on the number of segments that a program invariant - a modular equation about the number of segment merges - holds.

Let α be some configuration with k segments $s_1, s_2, \cdots, s_k$ clockwise arranged on the ring. Suppose that γ, γ' are two consecutive configurations not before α. If two adjacent segments $s_i(\gamma), s_j(\gamma)$ merge into one segment at configuration γ. We call the merged segment at configuration γ' $s_i(\gamma')$, where $i < j$. If $g_i(\gamma), g_j(\gamma)$ are gaps of $s_i(\gamma), s_j(\gamma)$ respectively, then the merged gap is

$$g_i(\gamma') \equiv g_i(\gamma) + g_j(\gamma) - 1 \pmod{n-1}$$

If the step $\gamma \rightarrow \gamma'$ is not a merging step, then

$$s_i(\gamma') = s_i(\gamma),\ g_i(\gamma') = g_i(\gamma) \quad \forall i$$

In the following definitions, P is a fixed processor.

Definition. A valid step of P is a step of Rule A taken by P and of the form
$a.t\ \underline{a.u} \rightarrow (a+1).0$. ∎

Definition. $M(\alpha, \gamma, s_i(\gamma))$ is the number of merges associated with segment $s_i(\gamma)$ between configurations α and γ with respect to process P, its value is defined by the following rules:

1. $M(\alpha, \alpha, s_i(\alpha)) = 0$ for $i = 1, 2, \cdots, k$
2. Let γ, γ' be two consecutive configurations not before α. Assume that P is in segment $s_l(\gamma')$ at configuration γ'.
   If $l = 1$ then $M(\alpha, \gamma', s_i(\gamma')) = 0$ for all i.
   If the step $\gamma \rightarrow \gamma'$ is not a merging step, then

   $$M(\alpha, \gamma', s_i(\gamma')) = M(\alpha, \gamma, s_i(\gamma)) \text{ for all existing } i.$$

   If the step $\gamma \rightarrow \gamma'$ merges two segments $s_i(\gamma)$ and $s_j(\gamma)$ to $s_i(\gamma')$ $(i < j)$, then

   $$M(\alpha, \gamma', s_h(\gamma')) = M(\alpha, \gamma, s_h(\gamma)) \text{ for } h \neq i, j.$$

   and

   $$M(\alpha, \gamma', s_i(\gamma')) = \begin{cases} M(\alpha, \gamma, s_i(\gamma)) & \text{if } i < l \\ M(\alpha, \gamma, s_i(\gamma)) + \\ M(\alpha, \gamma, s_j(\gamma)) + 1 & \text{if } i \geq l \end{cases}$$

   For convenience, we also define $\gamma' = \gamma + 1$.

∎

By the above definition, we know $M(\alpha, \gamma, s_i(\gamma)) = 0$ for $i < l$.

Lemma 9 *If there are k gaps $g_1, g_2, \cdots, g_k$ at some configuration, then $\sum_{i=1}^{k} g_i \equiv k - 1 \pmod{n-1}$.*

Proof. WLOG assume that k segments are clockwise ordered, and their lengths are $l_1, l_2, \cdots, l_k$ respectively. Let label a be the head of the first segment , then the head of last segment has label

$$b \equiv a + \sum_{i=1}^{k-1} (l_i + g_i - 1) \pmod{n-1}$$

and since $b + l_k + g_k - 1 \equiv a \pmod{n-1}$, we have $\sum_{i=1}^{k} (l_i + g_i - 1) \equiv 0 \pmod{n-1}$. Therefore

$$\sum_{i=1}^{k} g_i \equiv \sum_{i=1}^{k} l_i + k = n + k \equiv k - 1 \pmod{n-1}$$

∎

Lemma 10 *Suppose that processor P takes a sequence of n consecutive valid steps at configurations $\gamma_0, \gamma_1, \cdots, \gamma_{n-1}$, then the ring consists of one segment at configuration γ_{n-1}*

Proof. WLOG assume that at γ_1, there are k segments $s_1(\gamma_1), s_2(\gamma_1), \cdots, s_k(\gamma_1)$ clockwise arranged on the ring with P as the head of $s_1(\gamma_1)$, and k gaps are $g_1(\gamma_1), g_2(\gamma_1), \cdots, g_k(\gamma_1)$ respectively. Because there are at most $n-1$ segments left after the first valid step of P, we know that $k \leq n-1$. When P takes a valid step, it joins the next segment unless this valid step is a merging step. So that there exists a configuration γ_l $(l \leq n-1)$ such that P is about to join segment $s_1(\gamma_l + 1)$ for the first time after P left segment $s_1(\gamma_1)$.
Claim $\forall\ \gamma_1 < \gamma \leq \gamma_l$, if P is in segment $s_h(\gamma)$ at configuration γ and $h \neq 1$, then for all $j \geq h$

$$(*) \qquad g_j(\gamma) + M(\gamma_1, \gamma, s_j(\gamma)) \equiv 0 \pmod{n-1}$$

Proof of Claim (By induction on $i = \gamma - \gamma_1$)
Base Case: $i = 1$
$\gamma_1 \rightarrow \gamma$ is a valid step of P. If $h \geq 2$ then $h = k$, it implies that $\gamma_1 \rightarrow \gamma$ is not a merging step. Hence

$$g_k(\gamma) + M(\gamma_1, \gamma, s_k(\gamma)) = g_k(\gamma_1) + M(\gamma_1, \gamma_1, s_k(\gamma_1)) = 0$$

Inductive Hypothesis: Assume that (*) holds for $\gamma = \gamma_1 + i$ and P is in segment $s_h(\gamma)$.
Let P be in segment $s_{h'}(\gamma + 1)$, we are going to show that (*) holds for $\gamma + 1$ and h'. We have four cases.
Case 1 ($\gamma \rightarrow \gamma + 1$ is not a merging step) and ($h' = h$)
Nothing changes, (*) still holds.
Case 2 ($\gamma \rightarrow \gamma + 1$ is not a merging step) and ($h' < h$)
Because (*) holds for $i \geq h$, we only need to show that (*) holds for $i = h'$ too.
Since $h' < h$, $\gamma \rightarrow \gamma + 1$ is a valid step of P. Thus

$$g_{h'}(\gamma + 1) = g_{h'}(\gamma) = 0 \quad \text{and}$$

$$M(\gamma_1, \gamma+1, s_{h'}(\gamma+1)) = M(\gamma_1, \gamma, s_{h'}(\gamma)) = 0$$

Case 3 ($\gamma \rightarrow \gamma+1$ is a merging step) and ($h' = h$) Assume two segments $s_i(\gamma), s_j(\gamma)$ ($i < j$) are merged by this step. If $i \geq h$ then we know by inductive hypothesis that:

$$g_i(\gamma) + M(\gamma_1, \gamma, s_i(\gamma)) \equiv 0 \pmod{n-1}$$

$$g_j(\gamma) + M(\gamma_1, \gamma, s_j(\gamma)) \equiv 0 \pmod{n-1}$$

Therefore

$$\begin{aligned} & g_i(\gamma+1) + M(\gamma_1, \gamma+1, s_i(\gamma+1)) \\ \equiv\ & g_i(\gamma) + g_j(\gamma) - 1 + M(\gamma_1, \gamma, s_i(\gamma)) + \\ & M(\gamma_1, \gamma, s_j(\gamma)) + 1 \\ \equiv\ & 0 \pmod{n-1} \end{aligned}$$

Case 4 ($\gamma \rightarrow \gamma+1$ is a merging step) and ($h' < h$) $\gamma \rightarrow \gamma+1$ is a valid step of Pi, and $s_{h'}(\gamma), s_h(\gamma)$ are merged by this step. So that (*) holds for $i > h$ by inductive hypothesis. $g_{h'}(\gamma) = 0$ and $M(\gamma_1, \gamma, s_{h'}(\gamma)) = 0$ give us

$$\begin{aligned} & g_{h'}(\gamma+1) + M(\gamma_1, \gamma+1, s_{h'}(\gamma+1)) \\ \equiv\ & g_{h'}(\gamma) + g_h(\gamma) - 1 + M(\gamma_1, \gamma, s_{h'}(\gamma)) + \\ & M(\gamma_1, \gamma, s_h(\gamma)) + 1 \\ =\ & g_h(\gamma) + M(\gamma_1, \gamma, s_h(\gamma)) \\ \equiv\ & 0 \pmod{n-1} \end{aligned}$$

End proof of **Claim** ∎

We now prove the lemma by contradiction. Assume that there are $r \geq 2$ segments at configuration γ_l, and P is in segment $s_h(\gamma_l)$. By the definition of l, we know that segments $s_h(\gamma_l)$ and $s_1(\gamma_l)$ are adjacent. And because P is about to join segment $s_1(\gamma_l+1)$, $g_1(\gamma_l) = 0$. Recall that $M(\gamma_1, \gamma_l, s_1(\gamma_l)) = 0$. Applying **Claim** to γ_l, we obtain

$$g_j(\gamma_l) + M(\gamma_1, \gamma_l, s_j(\gamma_l)) \equiv 0 \pmod{n-1} \quad \text{for all } j$$

$$\Rightarrow \quad \sum_j g_j(\gamma_l) + \sum_j M(\gamma_1, \gamma_l, s_j(\gamma_l)) \equiv 0 \pmod{n-1}$$

By **Lemma 1**, $\sum_j g_j(\gamma_l) \equiv r-1 \pmod{n-1}$.

$$\Rightarrow \sum_j M(\gamma_1, \gamma_l, s_j(\gamma_l)) \equiv$$

$$-\sum_j g_j(\gamma_l) \equiv n-1-(r-1) \equiv n-r \pmod{n-1}$$

This implies that

$$\begin{aligned} & \text{Total number of merges between } \gamma_1 \text{ and } \gamma_l \\ \geq\ & \sum_j M(\gamma_1, \gamma_l, s_j(\gamma_l)) \\ \geq\ & n-r \end{aligned}$$

Therefore, there are at most $k-(n-r)$ segments left at configuration γ_l. But $k-(n-r) < r$, a contradiction. ∎

Theorem 3 *The protocol correctly detects the stability of ring.*

Proof. It is easy to see that once the system is stabilized, all counters will eventually reach the value 3. We are going to show that if a counter C has value 5, then the system is stable.

Assume that P has taken m consecutive steps of Rule A of the form $a.t\ \underline{a.u} \rightarrow (a+1).0$ when counter C reaches value 3, and that the *i-th* step has the form $a_i.t\ \underline{a_i.u} \rightarrow (a_i+1).0$, for $i = 1, 2, \cdots, m$. We then have $a_{i+1} = a_i + 1 \pmod{n-1}$ and the sequence $a_1, a_2, \cdots, a_m$ contains either two 0s and one $\frac{n-1}{2}$ or one 0 and two $\frac{n-1}{2}$s. This implies that $m \geq n$. By **Lemma 10**, the ring has only one segment in it now. Because Rule B does not pass a tag across a state with label 0, we know that the state 0 has tag 0 when counter C has value 4. When $C = 5$, all tags are 0, hence the ring is stable. ∎

6 The Delay of Detection

There is no protocol which detects the stability as soon as ring becomes stable. We study the amount of delay in this section. Note that we are interested on the delay after the ring becomes stable, so even though the ring is asynchronous, it makes sense to talk about cycles, since a single token is passed along the ring, and these are the only enabled transitions.

At a legitimate configuration, a processor P can have state 0 and its observer's counter C can have the value zero. P's state increases by one each cycle, counter C reaches 3 after $\frac{3}{2}(n-1)-1$ cycles. C will be 5 after two more cycles. Therefore, we have

Theorem 4 *The protocol of the previous section has a delay of $\frac{3n-1}{2}$ cycles.*

We can reduce the delay to $n+1$ cycles by a modified protocol that uses a $\lceil \lg(n+2) \rceil$ bit counter instead of a five bit one. The observer O initial its counter C to 0. If its processor P takes a step of Rule A of form $a.t\ \underline{b.u} \rightarrow (a+1).f(b-a)$. Then P sets

$$b_1 = \begin{cases} 1 & \text{if } b = a \\ 0 & \text{otherwise} \end{cases}$$

and sends b_1 to its observer O. After receiving b_1 from P, O increases its counter by one unless C has reached value $n+2$. The observer detects the stability when $C = n+2$. Compare this modified protocol with

the original protocol: we see that there is a trade-off between the delay and the space used by observers. This delay is unavoidable, as shown by the theorem below.

Theorem 5 *If the state of the observers is influenced only by the state of the associated processor and by the messages that pass through it, then any protocol that detects stability has a delay of at least n cycles (n^2 steps).*

Proof. Let us consider the initial configuration γ_0

$$0.0\ 0.0\ \cdots\ \underline{0.0}$$

The processor with an underlined state is the processor chosen by the demon to take a move and the rightmost processor is P. Let O be P's observer. A sequence of steps is as follows:

$$\begin{array}{ll}
0.0\ 0.0\ \cdots\ 0.0\ \underline{0.0}\ 1.0 & \\
0.0\ 0.0\ \cdots\ \underline{0.0}\ 1.0\ 1.0 & \\
\cdots & \\
\underline{0.0}\ 1.0\ \cdots\ 1.0\ 1.0 & \\
\underline{2.f(n-2)}\ 1.0\ \cdots\ 1.0\ 1.0 & \text{by Rule B} \\
2.0\ 1.0\ \cdots\ 1.0\ \underline{1.0} & [\gamma_1]
\end{array}$$

From γ_0 to γ_1, every processor takes a step of Rule A, that is a cycle. O has information $(0.0, 1.0)[\gamma_0 \rightarrow \gamma_1]$ at configuration γ_1. Similarly, the following configurations $\gamma_2, \ldots \gamma_{n-3}, \gamma_{n-2}, \gamma_{n-1}$ will be reached:

$$\begin{array}{l}
3.0\ 3.0\ 2.0\ \cdots\ 2.0\ \underline{2.0} \\
\cdots \\
\underline{(n-2).0\ \cdots\ (n-2).0\ (n-3).0\ (n-3).0\ (n-3).0} \\
0.0\ 0.0\ \cdots\ 0.0\ 0.f(n-2)\ (n-2).0\ \underline{(n-2).0} \\
1.0\ \underline{1.0}\ \cdots\ 1.0\ 1.f(n-2)\ 0.0
\end{array}$$

Again, from γ_i to $\gamma_{i+1}, i = 1, 2, \cdots, n-2$, every processor takes a step of Rule A. So the system takes $n-1$ cycles from γ_0 to γ_{n-1}. The system then takes moves by the following sequence

$$\begin{array}{ll}
1.0\ 2.0\ \underline{1.0}\ \cdots\ 1.0\ 1.f(n-2)\ 0.0 & \\
\cdots & \\
1.0\ 2.0\ \cdots\ (n-2).0\ \underline{1.f(n-2)}\ 0.0 & \\
1.0\ 2.0\ \cdots\ (n-2).0\ \underline{0.f(2)}\ \underline{0.0} & \\
\underline{1.0}\ 2.0\ \cdots\ (n-2).0\ 0.f(2)\ 1.0 & [\gamma_n]
\end{array}$$

The observer O has information

$$(**) \quad (0.0, 1.0, 2.0, \cdots, (n-1).0, 0.0, 1.0)[\gamma_0 \rightarrow \gamma_n]$$

at configuration γ_n.

Consider another initial configuration α_0

$$1.0\ 2.0\ \cdots\ (n-2).0\ 0.0\ \underline{0.0}$$

α_0 is a legitimate configuration. Again, we assume that P is the rightmost processor. The following configurations will be reached from α_0.

$$\begin{array}{ll}
2.0\ 3.0\ \cdots\ (n-2).0\ 0.0\ 1.0\ \underline{1.0} & [\alpha_1] \\
3.0\ 4.0\ \cdots\ 0.0\ 1.0\ 2.0\ \underline{2.0} & [\alpha_2] \\
\cdots & \\
1.0\ 2.0\ \cdots\ (n-2).0\ 0.0\ \underline{0.0} & [\alpha_{n-1}] \\
2.0\ 3.0\ \cdots\ (n-2).0\ 0.0\ 1.0\ \underline{1.0} & [\alpha_n]
\end{array}$$

At configuration α_n, P's observer O has information

$$(0.0, 1.0, 2.0, \cdots, (n-1).0, 0.0, 1.0)[\alpha_0 \rightarrow \alpha_n]$$

This information is exactly the same as $(**)$, thus O cannot make the detection at configuration α_n. Because the system takes n cycles from α_0 to α_n, any detecting protocol must have a delay of at least n cycles. ∎

There is one cycle difference on delays between modified protocol and **Theorem 3**. This is because we didn't use the information about tags in our protocol. If detecting protocols only use information about labels, then **Theorem 3** holds for $n+1$ cycles. To see this, note that the following configurations can be reached from γ_n

$$\begin{array}{l}
1.0\ 2.0\ \cdots\ (n-2).0\ 0.f(2)\ \underline{1.0} \\
[\gamma_n] \text{ By Rule B} \\
\underline{1.0}\ 2.0\ \cdots\ (n-2).0\ 0.f(2)\ 1.f(2) \\
\cdots \\
2.f(2)\ 3.f(2)\ \cdots\ (n-2).f(2)\ 0.0\ 1.0\ \underline{1.f(2)} \\
\underline{2.f(2)}\ 3.f(2)\ \cdots\ (n-2).f(2)\ 0.0\ 1.0\ 2.0
\end{array}$$

7 The Number of States Required by BP Type Protocols

As mentioned in [2], Seger has shown that any correct uniform unidirectional protocol for a ring of n processors must use at least $n-1$ states per processor. There is still a gap between the lower bound and the upper bound on the number of states per processor.

It is clear that a BP type protocol with a good function f will require $(n-1)|f(T)|$ states in each processor.

Two functions are given and proven to be good in [1]. One is $f(k) = k$, which gives a protocol with $(n-1)(n-2)$ states. The other function is $f(k) =$ *the smallest prime divisor of* $n-k$, it gives a protocol with $(n-1)\frac{n}{\ln n}$ states by the prime number theorem.

Our goal is to find a good function which has small range.

Let us consider a class $\mathcal{F}$ of functions from T to T:
$\mathcal{F} = \{h \mid h(k) = 0$ iff $k = n$, and if
$\exists\, 1 < k < n$ and $a_1, a_2, \cdots, a_k$ in $T - \{0\}$,
such that $\sum_{l=1}^{k} a_l = n$, then
$\exists\, 1 \le i, j \le k$ such that $h(a_i) \neq h(a_j)\}$

Lemma 11 *Let $h \in \mathcal{F}$. Define $f(k) = h(n-k)$. f is a good function. In other words, the protocol defined by f satisfies condition 1 [No Deadlock] of* **Definition 1**.

Lemma 12 *Let f be a function from T to T such that $f(k) = 0$ iff $k = 0$. If the protocol defined by f is correct, then $h \in \mathcal{F}$, where $h(k) = f(n-k)$.*

The two lemmas above show that the problem of finding good BP type protocols is equivalent to the following interesting mathematical problem:

Color all integers between 1 and $n-1$ with m colors, such that given any $1 < k < n$ and for any group of integers $\{a_1, a_2, \cdots, a_k\}$ colored with same color, $\sum_{j=1}^{k} a_j \neq n$. How many colors are enough? How many colors are necessary?

Define

$$h(k) = \begin{cases} k & \text{if } 1 \le k < \sqrt{n} \\ \lceil \sqrt{n} \rceil + i & \text{if } k \ge \sqrt{n} \text{ and} \\ & \frac{n}{i} \le k < \frac{n}{i-1}, i = 2, 3, \cdots, \lceil \sqrt{n} \rceil \\ 0 & \text{if } k = n \end{cases}$$

Theorem 6 $h \in \mathcal{F}$.

Proof. Suppose that $\sum_{j=1}^{k} a_j = n$, where $a_j \in [1, n]\ \forall j$, $1 < k < n$, and all $h(a_j)$ have same value t.

Case 1: $t \le \sqrt{n}$

$a_1 = a_2 = \cdots = a_k = t$. Thus t is divisor of n. But n is a prime.

Case 2: $t > \sqrt{n}$

By definition, we have $\frac{n}{i} \le a_j < \frac{n}{i-1}$ for all j. Because n is prime, we get $\frac{n}{i} < a_j < \frac{n}{i-1}$ for all j. Thus

$$k\frac{n}{i} < \sum_{j=1}^{k} a_j < k\frac{n}{i-1}$$

Since $\sum_{j=1}^{k} a_j = n$, we get $i - 1 < k < i$. This contradicts the fact that k is an integer. ∎

This yields a protocol with at most $2n^{1.5}$ states in each processor. We can do better, at some effort.

Theorem 7 *For sufficiently large n, there exists a BP type protocol which uses $O(n\sqrt{\frac{n}{\ln n \ln\ln n}})$ states in each processor.*

Proof. Let $\{p_1, p_2, \cdots, p_r\}$ be all distinct primes $\le \frac{1}{3}\ln n = x$, and $\{p_1, p_2, \cdots, p_s\}$ be all distinct primes $\le \sqrt{\frac{n \ln n}{\ln\ln n}}$, where $s > r$. By the prime number theorem,

$$s \le O(\frac{\sqrt{\frac{n\ln n}{\ln\ln n}}}{\frac{1}{2}(\ln n + \ln\ln n - \ln\ln\ln n)}) \le O(\sqrt{\frac{n}{\ln n \ln\ln n}})$$

Set $m = \prod_{i=1}^{r} p_i$, since $\sum_{p \le x} \ln p < \frac{6}{5}x$, we have

$$m < e^{\frac{6}{5}x} = e^{\frac{6}{5}\frac{1}{3}\ln n} = n^{\frac{2}{5}} < \sqrt{\frac{n\ln\ln n}{\ln n}}$$

Let $\phi(m)$ denote the number of positive integers $\le m$ and relatively prime to m. We then have

$$\frac{\phi(m)}{m} = \prod_{p \le x}(1 - \frac{1}{p}) < \frac{1}{\ln x} = \frac{1}{\ln\ln n - \ln 3} = O(\frac{1}{\ln\ln n})$$

Let $n \equiv n_0 \pmod m, n_0 < m$. Because n is prime, $\gcd(n_0, m) = 1$. For $i < m$, if $\gcd(i, m) = 1$, then $yi \equiv n_0 \pmod m$ has unique solution $y \pmod m$. Let us call this unique solution y_i, that is, $iy_i \equiv n_0 \pmod m$. Define

$$h(y) = \begin{cases} 0 & \text{if } y = n & (1 \\ t & t \le s, p_t \text{ is the smallest prime divisor of } y & (2 \\ i\lceil\sqrt{n}\rceil + j & \text{otherwise, if } y \equiv i \pmod m, \\ & i < m \text{ and} \\ & \frac{n}{jm+y_i} \le y < \frac{n}{\max(1,(j-1)m+y_i)}, \\ & 0 \le j \le \lceil \frac{\sqrt{n\ln\ln n}}{m\sqrt{\ln n}} \rceil + 1 & (3 \end{cases}$$

The function h is well defined for all integers $1 \le y \le n$. If $h(y)$ is not defined by (1) & (2) , then $\sqrt{\frac{n\ln n}{\ln\ln n}} < y < n$ and $\gcd(y, m) = 1$. Therefore,there exists unique pair (i, j) such that

$$y \equiv i \pmod m \ \&\ \frac{n}{jm + y_i} \le y < \frac{n}{\max(1, (j-1)m + y_i)}$$

because $\gcd(i, m) = 1$. Thus, we know the range of h is of size:

$$\begin{aligned} &\le 1 + s + \phi(m)(\frac{\sqrt{n\ln\ln n}}{m\sqrt{\ln n}} + 2) \\ &\le O(\sqrt{\frac{n}{\ln n \ln\ln n}}) + O(\frac{\phi(m)}{m}\sqrt{\frac{n\ln\ln n}{\ln n}}) \\ &= O(\sqrt{\frac{n}{\ln n \ln\ln n}}) \end{aligned}$$

We now show that $h \in \mathcal{F}$. Suppose that $\sum_{j=1}^{k} a_j = n$, where $a_j \in [1, n]\ \forall j$, $1 < k < n$, and all $h(a_j)$ have same value v.

Case 1: $v \leq \sqrt{n}$

p_v is a common prime divisor for all a_l. This implies that p_v is divisor of n, but n is a prime.

Case 2: $v = i\lceil\sqrt{n}\rceil + j$

Because $a_l \equiv i \pmod{m}$, $\forall l$, let $a_l = mb_l + i$, then $n = \sum_{l=1}^{k} a_l = m\sum_{l=1}^{k} b_l + ki$. Therefore, we obtain that $ki \equiv n_0 \pmod{m}$, that is $k \equiv y_i \pmod{m}$.

On the other hand, because all a_l cannot be the same, we have

$$\frac{kn}{jm + y_i} < n = \sum_{l=1}^{k} a_l < \frac{kn}{\max(1, (j-1)m + y_i)}$$

$$\max(1, (j-1)m + y_i) < k < jm + y_i$$

This contradicts that $k \equiv y_i \pmod{m}$. ∎

References

[1] Brown,G.M., Gouda,M.G., and Wu,C.-L. Token systems that self-stabilize. *IEEE Trans. Comput. (June 1989)*, pp. 845-852.

[2] Burns,J.E.,Pachl,J. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems. (April 1989)*, pp. 330-344.

[3] Chang,E.J.H., Gonnet,G.H., and Rotem,D. On the costs of self-stabilizing. *Inf. Process. Lett. 24(1987)*, pp. 311-316.

[4] Dijkstra,E.W. Self-stabilizing systems in spite of distributed control. *Commum. ACM 17, 11 (Nov. 1974)*, pp. 643-644.

[5] Dijkstra,E.W. Self-stabilizing systems in spite of distributed control (EWD 391). Reprinted in *Selected Writing on Computing: A personal Perspective*. Springer-Verlag, Berlin, 1982, pp.41-46.

[6] Dijkstra,E.W.
A belated proof of self-stabilization. *Distributed Comput. 1, 1(1986)*, pp. 5-6.

[7] Rosser, J.B., Schoenfeld, L. Approximate formulas for some functions of prime numbers. *Illinois Journal of Math. Vol 6(1962)* pp. 64-94.

Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors

Juan Alemany Edward W. Felten

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 U.S.A.

Abstract

This paper considers the implementation of non-blocking concurrent objects on shared-memory multiprocessors. Real multiprocessors have properties not present in theoretical models; these properties can be exploited to design non-blocking protocols that are more efficient in practice than those allowed by theoretical models. These new protocols rely on the operating system to take action when a thread of control is delayed during its non-blocking update. We illustrate the effectiveness of this approach by presenting two protocols that address factors hindering the performance of Herlihy's standard non-blocking protocol [Herlihy 90, Herlihy 91a]. These factors are: resources wasted by attempted non-blocking operations that fail, and the cost of data copying. We demonstrate the importance of these factors experimentally, and show how they can be reduced using protocols that rely on operating system support. To reduce the overhead of failing non-blocking operations, our first protocol maintains information about the utilization of the shared object; experiments show that this protocol performs better than the known alternatives. To reduce the cost of data copying, we introduce a second, optimistic protocol that avoids copying, except in the case when a thread of control is delayed during its attempted update.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, CCR-8703049, and CCR-9002891), the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and Apple Computer Company. Felten was supported in part by an AT&T Ph.D. Scholarship and a Mercury Seven Fellowship.

The authors may be contacted via email at *{juanito,felten}@cs.washington.edu*.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0125...$1.50

1 Introduction

Programmers of shared-memory multiprocessors typically use critical sections guarded by locks to ensure consistency of shared objects. Locks are well-understood and easily supported in hardware, and much research has gone into the development of locking protocols with low latency and high throughput [Anderson 90, Mellor-Crummey & Scott 91]. However, locks suffer from a number of problems, the most important of which arise from the interaction between locks and CPU scheduling. For example, if a thread of control holding a lock is delayed (say, due to a page fault or a processor preemption), no other thread may operate on the object protected by the lock. One thread's delay may prevent the entire parallel program from making progress until that thread is able to run again.

Lamport introduced lock-free synchronization [Lamport 77], a technique that allows parallel threads to ensure consistency of a shared object while avoiding the problems of locks. Lock-free protocols are attractive because they ensure that a thread that is delayed while updating a shared object does not prevent other threads from making progress. Herlihy proposed a specific methodology for implementing non-blocking shared objects [Herlihy 90], based on a preprocessor that transforms a sequential implementation of an object to an equivalent concurrent non-blocking implementation.

Although non-blocking synchronization is a promising idea, it has not been used much in practice. One reason is that non-blocking synchronization is based on atomic primitives that are not available on most parallel hardware. Bershad [Bershad 91b] recently showed that with appropriate operating system support, these primitives can be implemented efficiently in a standard instruction set. Because this work is not yet widely known, practical experience with non-blocking synchronization is limited.

As Bershad's work implies, there are important dif-

ferences between real systems and the theoretical models underlying standard non-blocking protocols. The most important difference is that in real systems, all events causing significant delay are visible to the operating system. This allows the design of non-blocking protocols that rely on the operating system to take corrective action whenever a thread experiences a delay. Operating system support enables a much richer variety of protocols for non-blocking synchronization; we present new protocols that offer significant performance improvements over existing non-blocking protocols.

We identify some performance problems with current non-blocking protocols, and suggest strategies to address them. Herlihy [Herlihy 91a] and Bershad [Bershad 91b] have pioneered work in this area, addressing the effects of contention for shared objects. We focus on the following two problems: performance degradation caused by updates that fail, and the cost of data copying. We propose a protocol to address each problem; both protocols rely on support from the operating system.

When several threads of control are simultaneously trying to update a shared object, only one can succeed — the others will accomplish nothing. Updates that fail use valuable computational resources and slow the progress of other threads doing useful work. Herlihy [Herlihy 91a] addressed this problem by proposing a policy that uses exponential backoff to alleviate contention for the shared object. We propose a general framework for addressing this problem; within this framework many specific policies are possible. We present one such policy, and demonstrate experimentally its performance advantage over the standard protocol and over exponential backoff.

Data copying is a major component of current non-blocking protocols. We present the result of a study [Beeck & LaMarca 91] showing that copying can cause significant performance degradation, and we propose an optimistic protocol for non-blocking objects that avoids copying in the common case when a thread is not delayed during its update.

2 Differences Between Theoretical Models and Real Systems

Existing bus-based, shared-memory multiprocessor systems differ in several ways from standard theoretical models of asynchronous shared-memory computers. There are two fundamental differences relevant to this paper: the use of threads rather than physical processors, and the predictable nature of delays.

Rather than programming processors directly, programmers express parallelism in terms of threads of control. Roughly speaking, each thread can be viewed as a virtual processor; a program's threads are multiplexed onto physical processors. As in the theoretical model, the threads communicate via shared memory, and synchronize by using mutual exclusion mechanisms such as locks and condition variables. The operating system kernel allocates physical processors (dynamically) between the parallel programs that are running. In turn, each program schedules its own threads onto its processors; this scheduling is typically done by a runtime system that is linked with the parallel program.

A more significant difference resides in the nature of delays experienced by processors in the theoretical model, and threads in real programs. In the asynchronous model, processors are assumed to experience arbitrary delays at arbitrary times. There is no way to tell how long a delay will last, or whether it will ever end. In fact, the failure of a processor is often modeled as an infinite delay.

In contrast, on real systems, all delays experienced by a thread can be divided into three classes:

1. *Short delays:* These delays are common, but have a duration of at most a few tens of clock cycles. Short delays are caused by events such as cache and translation-buffer misses, bus and memory contention, and timer interrupts. Programmers typically do not think about individual delays of this type, but rather model them as a uniform degradation in execution speed.

2. *Long delays:* In addition, threads suffer from delays of long duration, e.g. 100,000 clock cycles or more. These delays are caused by page faults, I/O operations, processor preemptions, and rescheduling of threads. As Bershad observed [Bershad 91b], all long delays are caused by operating system events, so the operating system is always aware when a long delay begins or ends.

3. *Infinite delays, or failures:* Real shared-memory multiprocessors are not robust against failures of hardware or critical software components. Such failures are considered catastrophic; those applications that need to recover from failures use external mechanisms such as checkpointing or transactional logging. In this paper, we will not concern ourselves with failures — instead, we will assume that fault-tolerance is handled at another level of the system.

To summarize, in real multiprocessor systems both short and infinite delays can be ignored. Only long de-

lays have a significant effect, and they are always known to the operating system.

3 Blocking vs. Non-blocking Synchronization in Real Systems

In the theoretical model, a non-blocking shared object guarantees that some processor will succeed in accessing the object within a bounded length of time. In real systems, long delays are bounded in length, and happen at most once per instruction; therefore, even the simplest locking protocols are *technically* non-blocking. This is not much consolation to the programmer: if a thread holding a lock experiences a long delay, no other thread may access the object protected by the lock. Although the delay is finite, the performance penalty is unacceptable in practice.

Ideally, we would like a guarantee that progress will be made in time significantly less than a long delay. Mutual exclusion mechanisms do not have this property, since the thread with exclusive access may suffer a long delay. Non-blocking protocols are less sensitive to delays than are mutual exclusion protocols, because one thread's delay does not prevent other threads from making progress. This motivates the use of non-blocking protocols on real systems.

An key feature of real systems is that the beginning and end of a long delay are always known to the operating system. This fact allows us to propose protocols that rely on the operating system to take some action when a long delay begins or ends. As a result, a much wider range of non-blocking protocols can be designed. For example, the non-blocking protocols we propose in sections 6 and 7 depend on such operating system support.

Bershad was the first to use operating-system mechanisms to support non-blocking synchronization; he implemented a non-blocking Compare-and-Swap operation on a multiprocessor that did not have a hardware Compare-and-Swap primitive [Bershad 91b]. Bershad's work allowed existing non-blocking protocols to be implemented on a wider range of machines.

The main contribution of our paper is to show how operating system support can be exploited to design a wider range of non-blocking protocols than those allowed by the theoretical model. Our protocols rely on the fact that the operating system can perform a variety of actions in response to delays. By exploiting this flexibility, we develop protocols which offer better performance in practice.

4 Existing Non-Blocking Protocols

Lamport introduced lock-free synchronization [Lamport 77], a technique that allows parallel threads to ensure consistency of a shared object without requiring mutual exclusion. Lock-free shared objects can be divided into *non-blocking* objects and *wait-free* objects [Herlihy 91b]. Non-blocking objects guarantee that *some* thread accessing the object will complete its operation in a fixed number of steps. Wait-free objects guarantee that *all* threads will complete their accesses to the object within a fixed number of steps. In this paper we consider only non-blocking objects.

Herlihy [Herlihy 88, Herlihy 91b] has shown that it is impossible to construct non-blocking implementations of arbitrary concurrent objects with any combination of atomic *read*, *write*, *fetch-and-op* and memory to memory *swap*. There are, however, *universal* atomic operations which are capable of implementing arbitrary non-blocking objects [Herlihy 91b]. The best-known of these universal primitives are *Compare-and-Swap*, and the combination of *Load-Linked* and *Store-Conditional*.

4.1 Herlihy's Methodology for Non-blocking Objects

Herlihy [Herlihy 90, Herlihy 91a] introduced a technique by which a preprocessor can transform an implementation of an arbitrary object into an equivalent non-blocking concurrent implementation. Threads operating on the non-blocking object follow the protocol illustrated by the pseudocode in Figure 1. In order to perform a non-blocking update of the shared object, a thread first acquires a pointer to the current version of the shared object and uses this pointer to make a private copy of this version. Then, the thread updates this private copy and attempts to install it as the new version of the shared object. The thread's non-blocking operation succeeds if no new version of the shared object has been installed since the thread began its operation. Otherwise, the thread's operation fails, and the thread tries its update again.

This protocol is based on atomic primitives we will call **take_snapshot** and **check_and_install**. Their specifications are as follows: **take_snapshot** returns a pointer to the current version of the shared object. **check_and_install** installs a new version of the shared object if and only if no new version has been installed since the caller's last **take_snapshot**. **check_and_install** returns *Success* if the new version was installed, and *Failure* otherwise. Different instructions can be used to implement **take_snapshot** and

```
private var
  private_version : POINTER TO Object;

procedure NonblockingUpdate(var shared_object : SharedObject) {
  var
    snapshot : POINTER TO Object;

  repeat {
    snapshot := take_snapshot(shared_object);
    copy object from *snapshot to *private_version;
    compute_new_value (private_version);
  } until (check_and_install (shared_object, private_version) = Success);
}
```

Figure 1: Pseudocode for Herlihy's standard non-blocking protocol. Private variables are kept on a per-thread basis. The '*' operator dereferences a pointer. Certain details of memory management are omitted.

check_and_install. Herlihy originally proposed using Load for **take_snapshot** and Compare-and-Swap [1] for **check_and_install** [Herlihy 90]. Later, Herlihy advocated using Load-Linked for **take_snapshot** and Store-Conditional for **check_and_install** [Herlihy 91a].

4.2 Implementation of take_snapshot and check_and_install

For concreteness, Figure 2 shows an implementation of **take_snapshot** and **check_and_install** based on timestamps. Each attempted update of the shared object is tagged with a timestamp; timestamps are unique and increasing in time. A shared variable *kill_timestamp* is maintained. When an update succeeds, it increases *kill_timestamp* to equal the highest timestamp given out so far; updates with timestamps less than or equal to *kill_timestamp* are operating on a stale version of the shared object and will fail.

[1] Compare-and-Swap does not exactly satisfy the specification for **check_and_install**. Compare-and-Swap checks whether the location *has the same value*, whereas **check_and_install** is required to check whether the location *has been modified*. It may be the case that the location has changed value from *A* to *B* (say), and then back to *A*. In this case, Compare-and-Swap will succeed when **check_and_install** would have failed. In Herlihy's protocol, **take_snapshot** and **check_and_install** are done on locations containing pointers to memory buffers; Compare-and-Swap may incorrectly succeed if memory buffers are recycled. Herlihy dealt with this problem by modifying the underlying buffer management protocol. The added complexity of this buffer management scheme is one reason Herlihy later favored Load-Linked/Store-Conditional over Compare-and-Swap.

4.3 Non-blocking Synchronization Without Hardware Support

Originally, the lack of hardware support for Compare-and-Swap or Load-Linked and Store-Conditional was a fundamental obstacle to using Herlihy's methodology. However, Bershad [Bershad 91b] observed that the lack of atomic instructions in hardware could be remedied given appropriate operating system support. He suggested software implementations using critical sections (guarded by locks) to atomically simulate Compare-and-Swap with "regular" instructions. When a thread inside one of these critical sections is delayed, the operating system notifies the runtime system which in turn backs the delayed thread out of the critical section or rolls it forward past the critical section [Bershad 91a]. Because of the special structure of the critical section implementing Compare-and-Swap, this roll-back or roll-forward is always possible. Bershad showed that the performance of his software implementation of Compare-and-Swap was acceptable: due to the small size of the critical section implementing Compare-and-Swap, threads were almost never delayed within it. Bershad's technique extends to other primitives including Load-Linked and Store-Conditional, and the timestamp-based implementations of **take_snapshot** and **check_and_install** shown in Figure 2.

5 Problems with Current Non-blocking Protocols

Although operating system support allows us to compensate for the lack of hardware support, a more funda-

```
ype SharedObject is
  current_version : POINTER TO Object;
  current_timestamp : Integer INITIALLY 0;
  kill_timestamp : Integer INITIALLY 0;
nd

rivate var
  my_timestamp : Integer;

rocedure take_snapshot(var obj : SharedObject){
  var
    snapshot : POINTER TO Object;

  atomically {
    snapshot := obj.current_version;
    increment obj.current_timestamp;
    my_timestamp := obj.current_timestamp;
  }
  return snapshot;
}

procedure check_and_install(var obj : SharedObject, new_version : POINTER TO Object){
  atomically {
    if (my_timestamp > obj.kill_timestamp){
      obj.current_version := new_version;
      obj.kill_timestamp := obj.current_timestamp;
      return Success;
    } else {
      return Failure;
    }
  }
}
```

Figure 2: Pseudocode for a timestamp-based implementation of **take_snapshot** and **check_and_install**. Private variables are kept on a per-thread basis.

mental problem with Herlihy's methodology is the overhead that results from having multiple threads compute a new version of the shared object. This overhead can be dissected into two components: useless parallelism and unnecessary copying.

Useless Parallelism Consumes Resources. Under the standard protocol, several threads may simultaneously attempt to update the shared object. Of all these threads only one will succeed in its update; all the other threads will fail and try again. Threads that fail are using computational resources that might be put to better use, for instance by running another thread on that processor.

Even worse, the failing threads use resources such as the memory bus when attempting their updates, degrading the performance of the thread that is successful. In short, the losers not only lose but they slow down the winner.

Unnecessary Copying Slows Down Updates. In Herlihy's methodology each thread makes a copy of the shared object and then performs a computation on this copy. This ensures that when any other thread examines the shared object, it sees it in a consistent state. The drawback is that this forces threads to copy the *whole* object even if they will modify only a small part. In cases where only small sections of the object are modified this will result in unnecessary copying which will slow down computation even further, especially if memory bandwidth is scarce.

6 Reducing Useless Parallelism

We can reduce the performance cost of failed updates by controlling the number of threads that simultaneously attempt updates to a single object. Excess threads can "get out of the way" by yielding their processor to another thread, or simply by waiting for the demand for the shared object to abate.

There are many possible policies to control the number of simultaneous update attempts. Of course, any policy must satisfy the non-blocking property — no thread can be prevented from beginning its update, unless some other thread is currently making progress on its update. The implementer is free to choose policies on a case-by-case basis, taking advantage of application-specific knowledge if available. Herlihy [Herlihy 91a] proposes one policy of this type, based on exponential backoff.

6.1 A Family of Protocols that Reduce Useless Parallelism

In this subsection we present a family of policies for reducing useless parallelism. These policies maintain a count of the number of updates to the shared object currently in progress. Threads defer their updates if this count is greater than some (usually small) constant K. Threads increment the count when they begin their update, and decrement the count when they finish their update. When a thread is delayed during its update, the runtime system decrements the counter so another thread may begin its update; this guarantees the nonblocking property. When the delayed thread is awakened, the runtime system increments the counter. The full pseudocode for our algorithm is shown in figure 3.

At first glance, this approach appears no different from conventional mutual exclusion. In fact, in the absence of delays, the algorithm behaves the same as mutual exclusion. However, if a thread is delayed while it is operating on its copy of the shared object, the operating system notices this delay, and notifies the runtime system, which in turn allows another thread to start an operation. A thread that is delayed does not prevent other threads from making progress; the delayed thread is working only on its private version of the object, and other threads still see a consistent public version. Our algorithm is non-blocking, while standard mutual exclusion protocols are not.

6.2 Performance Implications

To illustrate the performance improvement possible by reducing useless parallelism, we conducted an experiment on three non-blocking implementations of a 32-byte object. One implementation, called PLAIN, used Load-Linked and Store-Conditional, implemented in software using spinlocks. The second implementation, called BACKOFF, used exponential backoff as suggested by Herlihy [Herlihy 91a]. The final implementation, called SOLO, used the policy of section 6.1 with $K = 1$. We ran one thread on each processor; each thread was programmed to alternate between short periods of private computation and operations on the shared object. (The length of the private computation periods was chosen to be about half the time required for an attempted update of the shared object.) For each implementation, we measured the total number of successful non-blocking operations per second as a function of the number of processors. The measurements were taken on a Sequent Symmetry with 20 processors, and the results are shown in Figure 4.

The figure shows that the throughput of our protocol is higher than that of the plain protocol and the backoff protocol. All three protocols do the same amount of copying; the performance differences can be attributed to bus contention, which can be traced to two causes: contention for synchronization variables such as locks and the count of active threads, and failing updates' use of the bus to make their private copy of the shared object. PLAIN suffers from contention of both types. SOLO does not have any failing updates; its performance degrades as processors are added because of contention for the count of active threads. BACKOFF suffers from synchronization contention, from occasional failing updates, and from periods in which no thread is attempting an update but all waiting threads are "backed off".

The experiment fails to measure a real-life effect that favors our SOLO protocol over PLAIN and BACKOFF. In a practical situation, threads that defer their updates in the SOLO protocol could yield their processors to other threads, which could accomplish useful work. In the experiment, these threads simply waited for the count to decrease below K.

7 Reducing Unnecessary Copying

Recall that in current non-blocking protocols, every time a thread attempts to update a non-blocking object, it must copy the entire object. If the object is

```
const
  K : Integer;

type SharedObject is
  active_threads : Integer INITIALLY 0;
  current_version : POINTER TO Object;
  current_timestamp : Integer INITIALLY 0;
  kill_timestamp : Integer INITIALLY 0;
end

private var
  private_version : POINTER TO Object;
  snapshot : POINTER TO Object;
  my_timestamp : Integer;

procedure NonblockingUpdate(var obj : SharedObject){
  repeat {
    while(obj.active_threads >= K){ defer(); }
    snapshot := take_snapshot(obj);
    copy object from *snapshot to *private_version;
    compute_new_value (private_version);
  } until (check_and_install (obj, private_version) = Success);
}

procedure take_snapshot(var obj : SharedObject){
  atomically {
    increment obj.active_threads;
    snapshot := obj.current_version;
    increment obj.current_timestamp;
    my_timestamp := obj.current_timestamp;
  }
  return snapshot;
}

procedure check_and_install(var obj : SharedObject, new_version : POINTER TO Object){
  atomically {
    decrement obj.active_threads;
    if (my_timestamp > obj.kill_timestamp){
      obj.current_version := new_version;
      obj.kill_timestamp := obj.current_timestamp;
      return Success;
    } else {
      return Failure;
    }
  }
}
procedure ThreadDelayBegin_Notify(var obj : POINTER TO SharedObject){
  atomically decrement obj.active_threads;
}

procedure ThreadDelayEnd_Notify(var obj : POINTER TO SharedObject){
  atomically increment obj.active_threads;
}
```

Figure 3: Pseudocode for a non-blocking protocol that reduces useless parallelism. Private variables are kept on a per-thread basis. The '*' operator dereferences a pointer. The protocol maintains a variable, *active_threads*, that is equal to the number of threads actively executing updates. Threads entering the procedure NonblockingUpdate wait until *active_threads* is less than some constant K before trying their updates; during this waiting period the threads may spin, or they may yield their processor to another thread. The runtime system calls the procedure ThreadDelayBegin_Notify whenever a thread is delayed during its update; the procedure ThreadDelayEnd_Notify is called when the delay ends.

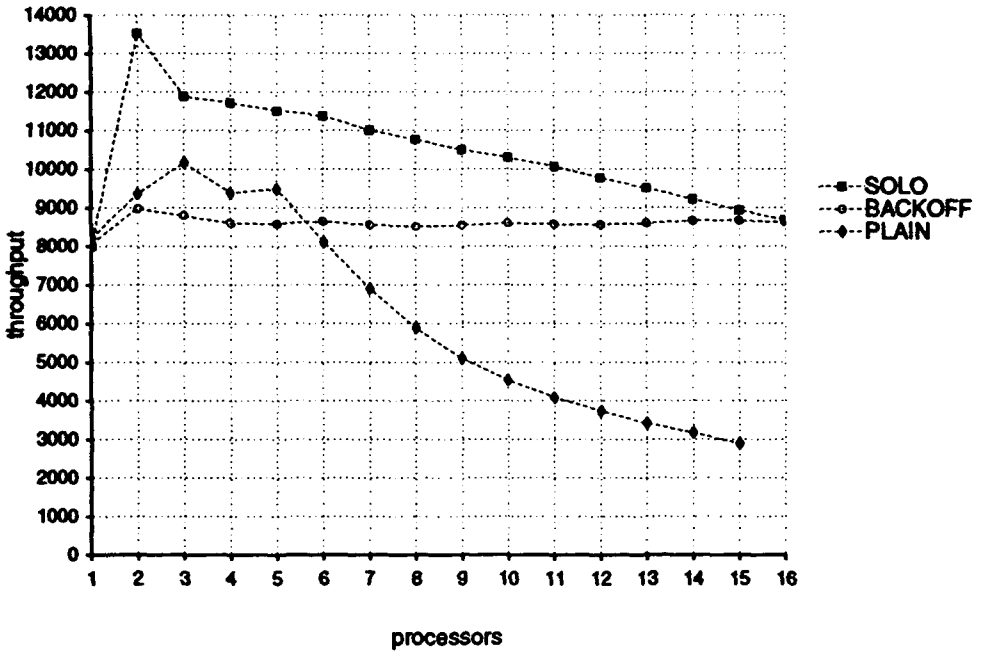


Figure 4: Throughput of three implementations of a 32-byte non-blocking shared object. Throughput is the number of successful non-blocking operations per second.

large, and the thread needs to modify a small part of it, copying the whole object is wasteful. This was illustrated by a study of non-blocking implementations of a name-server on our Sequent Symmetry [Beeck & LaMarca 91]. The implementations used Herlihy's protocol using *Load* for **take_snapshot** and *Compare-and-Swap* for **check_and_install**. The *Compare-and-Swap* primitive was implemented using a short critical section guarded by a lock.

This study found that the amount of copying involved in the non-blocking operation had a direct impact on performance, as shown in Figure 5. The authors originally followed Herlihy's approach and implemented the non-blocking protocol based on a sequential implementation of the name-server. They quickly realized that performance could be improved by reducing the amount of copying. Using optimizations specific to the semantics of the name-server, they were able to substantially reduce the amount of copying. Figure 5 shows the elapsed time required by different implementations of the name-server to perform 10080 operations. The three implementations differ in the amount of copying each thread must do to build its private copy of the shared object. The most optimized implementation required copying four bytes of memory.

7.1 An Optimistic Protocol to Reduce Copying

As shown in Figure 5, making a full copy of the shared object for each update carries a significant performance price. We would like to reduce the amount of copying that is necessary. We can do this by enlisting further support from the runtime system.

We present an optimistic protocol, which is designed to reduce copying in the common case when a thread is not delayed while updating the shared object. In the uncommon case when a thread is delayed during its update, the runtime system steps in and restores a consistent state.

The optimistic protocol is based on our protocol of section 6.1 with $K = 1$ (the SOLO protocol). This protocol allows only one thread to attempt an update at a time; all other threads either spin waiting for this thread to finish, or yield their processors to other threads. Only if a thread is delayed during its update is a second thread allowed to begin another update. When only one thread is carrying out an update, it can take advantage of this fact by working directly on the public version, rather than copying it. Of course, the thread must keep a log of its changes, so the runtime system can restore a consistent state if necessary. Additional threads will not notice that the object is in an inconsistent state, because their updates will not be allowed to start until the object is once again in a consistent state.

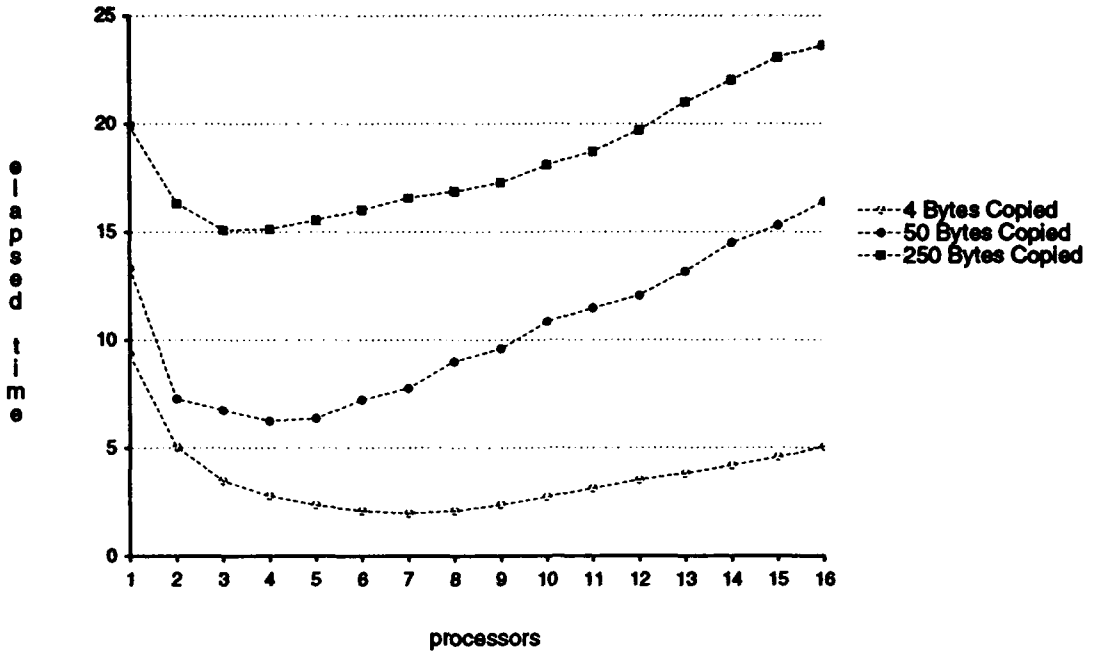


Figure 5: Elapsed time for 10080 operations in a non-blocking name-server. The three implementations differ only in the amount of data copied in each attempted non-blocking update.

When starting its update, a thread "borrows" the current public version of the shared object. It performs its changes directly on the borrowed version, maintaining a log. When the thread completes its update, it re-installs the borrowed version as the new "official" version, and discards its log.

If a thread is delayed during its update, the runtime system uses the thread's "dirty" borrowed version and the thread's log to build a consistent version of the shared object. This consistent version is identical to the version that the thread originally borrowed. The runtime system installs this reconstructed version as the current public version, and allows another thread to begin its update.

This second thread follows the same procedure: it borrows the current public version, and updates this borrowed version directly, while logging its changes. If this second thread is delayed, the runtime system will once again use the delayed thread's borrowed version and log to rebuild and install a consistent version of the shared object. A third thread is now allowed to begin its update. In this manner, an arbitrary number of threads can be delayed during their updates without preventing further threads from making progress. If several threads are working on updates simultaneously, whichever thread finishes first will succeed, and the other threads will fail.

7.2 Performance Tradeoffs

The optimistic protocol is best suited for those cases in which updates modify only a small portion of the shared object, and threads are rarely delayed during their updates. In these cases, the amortized cost of keeping the log and reconstructing the shared object is small relative to the cost of copying the whole object on every update.

On the other hand, if these assumptions are not true, a protocol such as SOLO will outperform the optimistic protocol. We note that it is possible to switch back and forth "on the fly" between the optimistic protocol and the SOLO protocol. The only difference is that threads borrow the public version in the optimistic protocol, and they copy the public version in the SOLO protocol. By switching between borrowing and copying, we can effectively switch between protocols. This allows the system to choose its protocol adaptively.

8 Future Work

Realistic implementation of the ideas presented in this paper requires support for an operating system mechanism such as scheduler activations [Anderson et al. 92]. We expect to have a version of Mach with scheduler activations running on our Sequent within the next few months. This will allow us to study the protocols in-

troduced in this paper in a more realistic setting. In particular, we would like to implement the optimistic protocol of section 7.1 and study its performance in real applications.

We would also like to see non-blocking synchronization made widely available. This requires an efficient implementation and an understanding of the required programming models and language support. For example, we envision a threads package that provides pervasive support for non-blocking operations. Widespread experience is necessary to seriously evaluate the practicality and convenience of non-blocking synchronization.

9 Conclusion

Non-blocking implementations of concurrent objects offer some important advantages over lock-based alternatives. In order to achieve the potential benefit of non-blocking synchronization, we must solve certain performance problems. Among these are the costs of useless parallelism and unnecessary data copying. We have demonstrated the effect of these problems experimentally, and suggested new protocols to improve the performance of non-blocking implementations.

Our protocols are based on features of real multiprocessor systems that are not present in the theoretical models for which non-blocking synchronization was developed. In real multiprocessors, events causing significant delays are visible to the operating system; our protocols take advantage of this fact by relying on the operating system to take corrective action whenever a thread is delayed. Operating system support allows our protocols to detect and react to delays; this added flexibility allows the design of more sophisticated protocols.

10 Acknowledgments

We are grateful to Richard Anderson, Ed Lazowska, Martin Tompa, Elizabeth Walkup, John Zahorjan, Maurice Herlihy and the members of the program committee for comments on earlier drafts of this paper. Langdon Beeck and Tony LaMarca generously shared the results of their non-blocking name-server study. Eric Koldinger and Raj Vaswani helped us understand the Sequent's memory subsystem. In addition, Brian Bershad, Hank Levy, Dylan McNamee, and Larry Ruzzo provided useful insights and ideas.

References

[Anderson 90] T. E. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[Anderson et al. 92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[Beeck & LaMarca 91] L. Beeck and A. LaMarca. A comparison of blocking and nonblocking synchronization. Class Project Report, Department of Computer Science and Engineering, University of Washington., 1991.

[Bershad 91a] B. N. Bershad. Mutual exclusion for uniprocessors. Technical Report CMU-CS-91-116, Carnegie Mellon University, 1991.

[Bershad 91b] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, 1991.

[Herlihy 88] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *7th ACM Symposium on Principles of Distributed Computing*, pages 276–291, 1988.

[Herlihy 90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 197–206, Mar. 1990. An updated and expanded version appears as [Herlihy 91a].

[Herlihy 91a] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Lab, 1991.

[Herlihy 91b] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991. A preliminary version appeared as [Her88].

[Lamport 77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, Nov. 1977.

[Mellor-Crummey & Scott 91] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

Robust, Distributed References and Acyclic Garbage Collection

Marc Shapiro
Peter Dickman
David Plainfossé

Projet SOR, INRIA, BP 105, 78153 Roquencourt Cédex, France

email: marc.shapiro@inria.fr

Abstract

We propose efficient, programming language-independent, location-transparent references as a substitute for pointers in distributed applications. These references provide the semantics of normal pointers for both local and distributed, transient and persistent objects. They may be passed in messages between and within nodes using a low-overhead presentation-layer protocol. The programmer remains free to create, delete or migrate objects at will. Sending references (or migrating objects) may cause references to be chained together across any number of spaces; we provide a short-circuit protocol to optimize access through such chains.

Integrated with these references, we provide efficient, distributed, garbage collection of acyclic data structures. Even in the presence of network failures such as lost messages, duplicated messages, out of order messages and site failures, the correctness of GC is guaranteed. The protocol assumes the existence of local garbage collectors of the tracing family. The protocol combines: (i) local tracing (from a conservative root); (ii) conservative distributed reference counting; (iii) periodic tightening of the counts; and (iv) allowance for messages in transit during GC. The protocol uses only information local to each site, or exchanged between pairs of sites; no global mechanism is necessary. It is parallel and should scale to very large systems, e.g. tens of thousands of nodes connected using both local and wide-area networks.

1 Introduction

Distributed object-based systems are of ever increasing importance, providing a powerful means of exploiting contemporary hardware. Writing distributed applications is, however, rarely as straightforward as writing local ones. A key reason is that local and remote objects are generally handled differently: a plain pointer may be used when a local object is created; a special handle must be used for remote objects. Code must therefore be aware of which kind of object it manipulates, and may need to be replicated to handle both kinds of objects. The solution to this problem is to provide a uniform programming model for both kinds of objects, that is, to provide a location-transparent reference mechanism.

We propose *references* as a substitute for pointers to objects in distributed applications. References provide uniform identification and access to local and remote objects. Methods of the target object of a reference can be invoked; references can be passed as arguments in invocations, both within and across spaces. Invocation of a referenced local object turns into a procedure call through a pointer. For remote objects, the invocation causes the parameters to be marshalled and sent to the object in remote procedure call (RPC) messages.

Intimately associated with our references is a fault-tolerant protocol for the detection of acyclic distributed garbage.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0135...$1.50

Some programming languages provide garbage collection (GC) to automatically deallocate inaccessible objects. GC is extremely useful, as it simplifies the programming model, therefore freeing valuable programmer time, while avoiding bugs and memory leaks which are notoriously hard to diagnose. However, there has been relatively little work on GC in systems supporting distribution. System designers often dismiss GC, viewed as too complex and/or too costly or just not useful for general applications. They are wrong on all counts. The bulk of distributed GC can be implemented in a generic, language-independent, low-overhead, fault-tolerant manner.

Our GC protocol is inexpensive in that it requires no extra foreground messages or system activity; furthermore, neither additional copying nor additional interpretation of message contents is required. The mechanism is safe in that as long as one reference to an object exists somewhere in the distributed system, the object's storage will not be reclaimed. We minimize the administrative message overhead by piggy-backing and batching. The protocol is fully parallel and scales well: the space complexity is proportional to the number of remote references; it communicates only between pairs of spaces; and third-party dependencies are avoided. As the only information used is local, or exchanged between pairs-of-sites, no global state or synchronization is necessary and neither multicast nor ordered protocols are required.

The underlying assumptions are weak and reasonable. Messages that arrive are delivered in finite, non-zero time; they may be lost, delivered out of order, or duplicated. The network may become partitioned. Nodes may crash silently. Clocks need not be synchronized. Local activities are not synchronized together.

When application code (the *mutator* processes), local garbage collectors, and a distributed collector all execute in parallel, it becomes difficult to guarantee consistency. Some published distributed GC algorithms assume strong consistency. In contrast, our design is based on weakening these assumptions. Strict consistency comes at a high cost; allowing "safe" inconsistency has the potential of greater efficiency, reliability and availability. For example, our protocol permits a space to infer that a remote reference to a local object exists when, in fact, there is no such reference. Other apparent inconsistencies may arise due to messages being in transit. Inconsistencies which do not violate the safety invariants of GC are harmless. Our mechanism relies on this fact, relaxing the conditions that usually guarantee liveness whilst maintaining safety invariants. This relaxation (as opposed to weakening) permits garbage to accumulate in the system. Tightening the conditions then directly results in the reclamation of the garbage. This process of relaxing and retightening is embodied in a straightforward windowing algorithm based on unsynchronised timestamps.

This paper presents our protocol abstractly. Section 2 introduces our model and Section 3 then provides a detailed description of the algorithms and the underlying data structures. After that, Section 4 analyzes the complexity of the protocol in terms of messages, time and space overhead. We compare this protocol to several others in Section 5.

In this presentation certain key features of the mechanism are highlighted and discussed in some detail. We omit the protocol for migrating objects, support for the deletion of non-garbage objects, and persistence mechanisms; these are described elsewhere [25]. We do not address the collection of cyclic distributed garbage here; see however Section 5.

2 Overview

The distributed universe of objects is subdivided into disjoint *spaces*.[1] It is assumed that a space can be identified unambiguously, e.g. by a UID[2]. A space has two possible states. It may be operating and communicating normally; or it may terminate. If a space terminates it does not reappear and its name is never reused. If the hardware it resides on crashes, a space may either persist (recover) or terminate. A space may also appear to cease communicating (*disconnect*), due to network problems, for example, or during temporary overload or recovery after a crash; eventually, however, such a space either recovers (*reconnects*) or terminates. In the case of a crash, our model does not specify whether the affected space(s) recover or terminate, nor how such termination is notified; one could postulate the existence of an external "oracle".

We assume that each space executes a local garbage collector (LGC) of the tracing family[3]. An LGC executes independently of the activity of other LGCs and of distributed cleanup.

Each space A carries a timestamp generator $stamp_A()$. The timestamp generators need not be synchronized across spaces.

Finally, each space also carries an array $threshold_A$

[1] We use the abstract term "space", rather than, for instance, "host", "node" or "process", to avoid committing to a particular implementation or lifetime.

[2] A higher level of distributed GC might be able to ensure that a space name is not reused until no further references to the old space exist; this is beyond the scope of this paper. Therefore we assume that all space names are unique.

[3] Such LGCs are standard in Lisp, Smalltalk, Eiffel and similar environments. LGCs are also being developed which are either language independent [5] or adapted to C and C++ [1, 9]. Tracing GC is fault tolerant, in that each execution of the collector is independent of all previous ones.

of timestamp values received from other spaces and limiting acceptable messages.

2.1 Objects and References

Spaces contain passive objects consisting of instance data and associated methods. An object's instance data may include any number of references to other objects. A reference is a location-transparent handle, through which methods of the target object may be invoked. A reference may be passed as an argument and thus copied between spaces. Our model is illustrated by Figure 1; which all the examples in this text refer to.

Inter-space references are identified in special data structures in the source and destination spaces. A space maintains a table of *stubs* for its outgoing remote references. There is never more than one stub in a space for a given object, even if that space contains many references to that object.

Complementing the stubs, each space maintains a table of *scions*,[4] which track incoming remote references. Every stub that points to a given space has a corresponding scion in that space. A scion may either point to a local object or a further stub; this permits chaining of remote references. Scions are conservative in that the stub corresponding to a scion may no longer exist. This inconsistency does not lead to errors, but may temporarily prevent some garbage from being reclaimed. Note that we use a distinct scion for each stub, unlike other systems in which multiple 'outgoing' pointers will merge into a single 'incoming' pointer.

A stub contains a *locator* composed of two parts, called strong and weak. Each part indicates a scion and consists of an identifier of a space containing the scion, and the scion's name valid within that space. The strong part identifies the scion which matches the containing stub. Distributed garbage collection relies on the invariant that (in the absence of failures) there is always an uninterrupted chain of stubs' strong parts and scions (hereafter called a "strong chain") between the source and target of a remote reference. The weak part identifies a scion which, while part of the same chain of strong indicators, may be closer to the target object[5]. Weak parts are used for communication and location, which rely on the invariant that the indicated scion will not be collected, being protected by the strong chain.

Mutators send and receive messages using a low-overhead "presentation-layer" protocol, with scions and stubs created or updated automatically as needed. The marshalled form of a reference is a locator.

[4] **Scion** *n.* 1. A descendant or heir. 2. A detached shoot or twig containing buds from a woody plant and used in grafting [16].

[5] Indeed, if the target object does not migrate, the weak part is guaranteed to point to its space of residence.

2.2 Garbage Collection

In discussing garbage collection we distinguish the *mutator* and *collector* rôles [8]. Garbage collection poses three distinct problems: distinguishing references from other data in objects; given these references, detecting garbage objects; and disposing of garbage objects, according to their semantics. The former and latter problem are language-dependent and are delegated to the local garbage collectors (LGCs), as is the detection of local garbage. Distributed detection is independent of object structure or semantics and is performed by our protocols. During local garbage collection of space A, the local collector's root set is augmented to consist of the local roots (noted R_A) and the local scions.

As noted in the introduction, a scion may exist for which the corresponding stub no longer exists. This may cause the local garbage collector to believe that there is a remote reference to a local object, whereas, in fact, none exists. For this reason, garbage collection is somewhat conservative. The protocols ensure that eventually the unreferenced scion will be reclaimed. Then, the objects reachable only from that scion become garbage and may be reclaimed by the local garbage collector. In our scheme, all unreferenced scions (i.e. unreachable and not on a distributed cycle of garbage) will eventually be reclaimed (to the extent that the LGC is itself exhaustive).

3 Specification of the Protocol

We distinguish four aspects of the mechanism and highlight novel features for each. Together, these offer both robust distributed references and the automated collection of acyclic distributed garbage:

- The Transport Protocol describes the way in which messages are handled (under what circumstances is a message discarded, for example).
- The Presentation Protocol describes the way in which references are marshalled into, and unmarshalled from, messages.
- The Invocation Protocol details the way in which a reference is used. That is, how locating of the target object interacts with the activity of invoking its methods.
- The Cleanup Protocol covers the elimination of data structures associated with garbage remote references.

When a space terminates, rather complex recovery behaviour may be required of other spaces. Rather than including details of this in each of the four protocols listed above, a separate section addresses Termination Recovery.

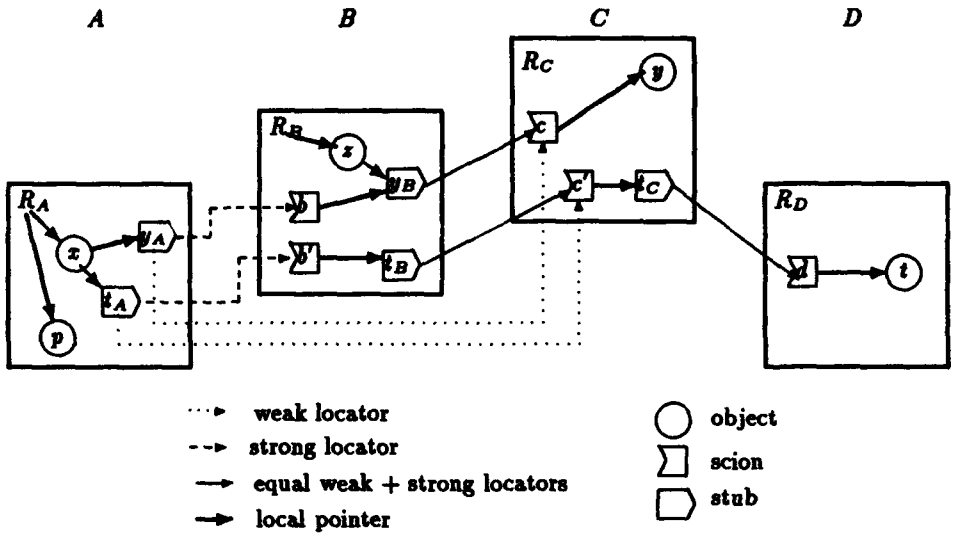


Figure 1: Object and reference model

3.1 Data structures

These mechanisms manipulate data structures of three key forms: scions, locators and stubs. Suppose y is located in space C; when used in B, a reference to y relies on a stub y_B in B and a scion c in C. The stub contains a *locator* $\{C, c, C, c\}$ with *strong* and *weak* parts, both of which, in this simple case, indicate scion c on space C. Scion c itself holds a pointer to y.

Scions contain a space-identifier (indicating the single remote space which potentially contains the matching stub), a locally generated timestamp (produced when a reference which relies on this scion was last locally marshalled), a pointer to the object (or to a stub if the object is remote). Scions are accessed in four modes: (i) for invocation and location, an individual scion is accessed directly, via a scion name included in the weak location of a stub; (ii) by enumeration of the local scions that are associated with a given remote space; (iii) by enumeration of local scions that point at some particular local object or stub; or (iv) by enumeration of all local scions. The enumeration modes are used, respectively, by the Cleanup Protocol, by reference marshalling, and by the local garbage collector. The scion data structure is documented in Table 1.

Locators are the marshalled form of references and are also the primary components of stubs. Locators are documented in Table 2.

Stubs contain a locator and a timestamp. Stubs are accessed in three ways: (i) invocation proceeds directly, through a local pointer to the stub; (ii) when a reference is unmarshalled from a message, it is compared against existing stubs (for unicity, and in case update of the weak part is needed) by strong locator; (iii) the Cleanup Protocol enumerates all local stubs containing a strong part that point at a given remote space. Stubs are documented in Table 3.

Table 1: Scion Data Structure

source_space: space_name	the name of the space holding matching stub.
target_object: pointer	a pointer to the object (or a stub if the target is not local).
stamp: timestamp	a locally-generated timestamp to protect in-transit references

Table 2: Locator Data Structure

strong_space: space_name	Space where next scion in chain resides
strong_scion: scion_name	Scion's name
weak_space: space_name	A space closer to where the object resides
weak_scion: scion_name	Scion's name

A locator contains both strong and weak parts. Unlike most similar mechanisms, both parts do lead (possibly indirectly) to the target of the reference, without any global search. There is always an uninterrupted chain, embodied in the strong locator parts, of stubs and scions from primary source to ultimate destination. The proof that the garbage collector is safe relies on this invariant.

Finally, each space A contains a table of received timestamps, $threshold_A$, indexed by space identifier, ex-

Table 3: Stub Data Structure

location: locator	Locator for the target object
stamp: timestamp	Timestamp to guard against race conditions

plained in the next section.

3.2 Transport Protocol

Communication between mutators in different spaces occurs via messages that are timestamped using the timestamp generator, $stamp_B()$, of the message's source B.

At the destination A, the message timestamp is compared with the B timestamp held in $threshold_A[B]$: only messages containing timestamps generated later than the threshold are accepted. This eliminates a race condition explained in Section 3.5.1.

Our approach usually permits messages to be processed out-of-order. Whenever message ordering could be critical, the required synchronization is explicitly supported using either call-response message pairs or the threshold table. Some delayed messages might be treated as lost, but only in circumstances in which it is possible that acting upon them would violate the GC invariants.

As our mechanisms are designed to tolerate message loss, reordering and duplication, it is acceptable for the message-passing protocol to use cheap, unreliable transport protocols. If an application uses a more reliable protocol, this causes no difficulties as the necessary conditions for this scheme are retained. The background messages required by the cleanup protocol may also use an unreliable protocol, even when the application itself requires a more reliable mechanism.

3.3 Presentation Protocol

A side-effect of marshalling a reference into a message is to produce a scion. There can only be a single scion per target object and per referring space. The marshalling code first searches for such a scion; if none exists it is created; it is timestamped with the current timestamp.

The marshalled form of a reference is a locator, the *strong* part of which names a scion in the sender's space. The *weak* part names a scion closer to the object (but on the same strong chain), if the sender knows one; otherwise it is equal to the strong part. These parts both permit the corresponding remote scion to be unambiguously identified. The message is timestamped with the same timestamp as used to create the scion(s) it refers to.

At the receiver, unmarshalling the reference produces a pointer to a single local stub per matching scion. The actions are: search for a stub with the same strong locator; if one exists and its timestamp is less than the message's, then copy the weak part and the timestamp from the message (if none exists, create one from the weak part and the timestamp in the message); pass up the address of this stub to the application.

Since marshalling and unmarshalling are necessary for remote communication, our approach adds negligible overhead other than creation of stubs and scions, while ensuring no duplicates. Care is taken to ensure uniqueness of the stub-scion pair referring to a particular object between two spaces. This has a small associated cost (discussed in Section 4.2.2), because it requires additional indexing mechanisms and searches, but it renders scion and stub creation idempotent. Hence, deletion of a stub permits the corresponding scion to be discarded without fear that another stub may depend on that scion.

The construction of both stubs and scions is conservative. The endpoints of the remote reference are created without knowing whether they will be useful, and stubs are always created after their matching scions. For instance, it may occur that a message containing a reference is lost; in this case, a scion has been created without a corresponding stub. It may also occur that a received reference is actually ignored by the mutator; in this case the whole reference chain is useless. The stub and scion code can only add new stubs and create more scions. This is consistent with the view that mutators only allocate objects, whereas deallocation is performed transparently by the collector. Here the LGCs remove unreferenced stubs, and the cleanup protocol removes unreferenced scions.

3.4 Invocations and Short-Circuiting Indirect References

A reference is typically used to invoke some procedure (or *method*) of the target object. Remote invocation uses a call-response protocol[6].

3.4.1 Indirect References

Liberal use of indirect reference chains allows a reference to be passed cheaply in messages, while retaining useful invariants (i.e. the guarantee of reachability). But, when considering communication, they are harmful: not only because of the overhead and poor locality, but more fundamentally because sending a reference along an indirect chain creates yet another indirect chain.

[6] Our initial specification [25] was based on one-way messages. A call-response protocol considerably simplifies short-circuiting indirect references.

Consider, for example, an invocation on y, made by x, passing a reference to y itself as an argument. On receipt of the invocation at C, assuming the strong locator chain was used, the argument will be indicated by a chain of stub-scion pairs starting in C and running through B, A, B again and back to C. If this same reference is returned as a result, matters deteriorate further.

3.4.2 Short-Circuiting an Indirect Reference

For these reasons, the weak locator chain is used for invocations[7] and the strong chain is lazily short-circuited in a safe fashion as a side-effect of such invocations. Obsolete indirect stubs and scions will be cleaned up later by the garbage collector. Furthermore, an indirect chain is short-circuited, at the latest before the harm indicated above may occur, i.e. before allowing execution of an invocation carrying reference arguments.

There are two sub-cases to consider. The easiest case is when the caller's weak locator is exact, indicating the scion closest to the target. The other case is when the harmful effect indicated above could occur (the weak locator is inexact, and at least one argument is a reference). The intermediate case (inexact weak locator but no reference argument) can be treated in either way.

3.4.3 Weak Locator Exact

In Figure 1, suppose x calls y.f(), i.e. invokes some method of y with no argument. The call message is sent to weak location $\{C, c\}$. Upon receipt of a call, at scion c, from space A other than the space containing the matching stub (B), a new scion c'' (which does not appear in the figure) is created. Locator $\{C, c'', C, c''\}$ is piggybacked on the invocation results, and the stub y_A at the invoker's space A, through which the invocation was made, is updated to locator $\{C, c'', C, c''\}$. As the original chain: $y_A \rightarrow \{b, B\} \rightarrow y_B \rightarrow \{c, C\} \rightarrow y$ remains in place until the y_A stub contents are changed, the GC invariants are maintained. The superseded indirect chain (the scion $\{B, b\}$ and, possibly[8], the stub y_B and the scion $\{C, c\}$) becomes garbage and will be collected at some later time.

3.4.4 Weak Locator Inexact

Consider now x invoking $t.\mathbf{g}(p)$, some method **g** of t with a reference argument (for which a scion a is allocated). Since the weak locator indicates C as above, the call message is similar.

Scion c', receiving this message, detects that x is not local because it points to a stub. The message is passed (without unmarshalling) to t_C, which forwards it on to its own weak locator, i.e. $\{D, d\}$. Here we notice that the target is local but that the caller's weak locator was inexact and the argument is a reference. Therefore the call is aborted and a location_exception is signalled back to A with an up-to-date location. For this, a new scion $\{D, d'\}$, pointing to t, is allocated for use by A.

Upon receiving the exception, t_A updates its locator to point to $\{D, d'\}$. It then retries the invocation (a new scion a' pointing to p on behalf of D must be allocated). Now scion $\{A, a\}$ can be collected, as well as the old indirection chain $b' \rightarrow t_B \rightarrow c' \rightarrow t_C \rightarrow d$.

3.5 Collector Protocol

Above we have specified the *mutator protocol*. Now we will specify the *collector*, i.e. actions performed independently of the mutator's execution, in order to collect garbage. This involves two independent activities: local garbage collection and the distributed cleanup protocol. Reclamation of unreachable stubs and scions is tricky, because of the possibility of lost messages, and of race conditions.

3.5.1 Local Garbage Collection

The LGC traces references from the local root and the set of all local scions. An unreachable stub is garbage, and can be collected. However there is a possible race condition with messages, containing the same reference, arriving late.

The race condition is eliminated by the following rule. Before discarding a stub in A, the strong locator of which points to space B, $threshold_A[B]$ is increased to the value in the stub's timestamp, causing the transport protocol at A to drop earlier messages from B.

3.5.2 Cleanup Protocol

As was noted in the introduction, the mutator protocol relaxes the GC liveness condition, and the cleanup protocol occasionally strengthens it: the strongest form is that every scion has a single matching stub, the weakened form permits some scions to have no matching stub.

Signaling to a scion that the matching stub has been collected is complicated by two potential problems. First, the "deletion" message could be lost. To tolerate message loss, lists of stubs which were still live at some time are sent, rather than sending deletion messages. Second, messages are asynchronous, leading to possible race conditions between scion update and deletion. To avoid the race condition, a scion is removed only if it is both unreachable and there is no message in transit which may make it reachable again. (This race condition is different from the one in the previous section.)

[7] If objects do not migrate, the weak locator is guaranteed to indicate the space containing the target object.

[8] Depending on whether they also form part of other, extant, references at B.

Thus a space A will periodically send to some other space B a message (called a live message) containing: (i) the list of scion names, taken from the strong locators of all extant stubs at A that point at scions at B, and (ii) the value $threshold_A[B]$.

This permits the receiver B to deduce what scions in B are unreachable: precisely those for which there is no matching stub. An unreachable B scion can be removed, if and only if the following condition holds for it at B:

$$scion.\mathsf{stamp} \leq message.\mathsf{threshold}$$

i.e. there are no recent messages in transit carrying its location, which could make it reachable again.

Essentially, we have made stub and scion deletion an idempotent operation and eliminated the race conditions by ignoring messages arriving "too late" and by never discarding scions that have recently updated timestamps. Scion timestamps protect against deletion of scions for which a usable reference may be in transit, whereas stub timestamps protect against re-creation of stubs for which scions have been discarded.

To ensure that progress is made, one space may prompt another to report on the stubs it holds. A space B may send a background prompt message to another space A. On receipt of such a message the live message is sent and processed as indicated above.

3.6 Termination Recovery

In this section the problems arising when a space terminates, are addressed. We define space termination to mean that its local root is deleted, as well as all objects it contains. (References to the deleted objects are detectably dangling; an attempt to invoke the target will raise an exception.) Indirection chains through the terminating space must first be resolved and short-circuited. These rules are easy to enforce when a space voluntarily terminates itself; in other cases a protocol is needed to achieve the same observable effect. There are two aspects: ensuring communication, and re-establishing the invariants.

3.6.1 Recovering Communication

Invocation uses the weak locator of the sender's stub, and therefore is not impaired by a break in the strong chain only. In fact, if the target does not migrate, the weak locator holds its actual location.

If objects do migrate, then the weak locator may point to an intermediate scion, such as t_A to c' in Figure 1. If the weakly located scion is lost, this also breaks the strong locator chain. Here the only possibility is to search exhaustively for the object. Such a search is expensive and may be prohibitive in large systems, so a structuring of the system into collections of spaces with intervening non-terminating gateways would be required.

Furthermore, global search assumes that the holder of a stub knows some unique feature of the sought-after object. Since we don't assume UIDs, the unique feature will be the weak locator part. Thus when an object migrates, its new scion must carry with it the list of scion names under which it had been previously known. This list will be discarded as a side-effect of discarding the scion in the short-circuit protocol of Section 3.4.2.

3.6.2 Re-Establishing the Invariants

Initially it might appear that termination of a space that contains a stub is of little consequence. Unfortunately, it is an error to simply discard the matching scion.

The safety of garbage collection depends on the invariant that an uninterrupted strong chain exists between the source and target of a reference. In turn, the invocation protocol depends on the fact that the scion pointed by a weak locator will not be collected.

We are contemplating a number of possible solutions. The simplest is to retain forever all scions whose source_space has not reliably short-circuited indirections through it.

Our preferred solution improves over the above, by relying on the existence of a global garbage collector (which is necessary anyway to collect distributed cycles of garbage, since they are not removed by the protocol presented in this paper) to detect and remove garbage scions retained in this way. This is the approach taken by Dickman [7] and means that the algorithm is no longer live, but remains efficient and effective.

An alternative would be to apply a rule, similar to the rule for objects, to broken chains: any reference chain indirecting through a terminated space is deemed dangling. Such an approach is only correct, however, if scion identifiers are never reused, as otherwise a different problem of erroneous chain following is introduced. It has the drawback that an object may become unreachable by some path, which happened to go through a terminated space, and remain reachable by others: such inconsistencies are undesirable.

Yet another solution maintains liveness and avoids further errors, but is expensive, involving a large-scale search. On discovering such a stub-less scion a message can be passed down the remaining chain to the object concerned[9]. The message has inserted in it the space and scion identifiers for every scion encountered during the message's journey. Having thus collected a list of all scions that may be indicated by detached sections of the

[9] If the chain is broken in two places the mid-section will be recovered first and this will then recover the association with the most detached part.

chain, an exhaustive search is performed. Each space in turn is presented with the list of intermediate scions and requested to update any and all relevent locators. Again the cost of global searches should be limited by a hierarchic structuring of spaces.

Whatever solution is chosen, the window of vulnerability can be narrowed by aggressively short-circuiting indirections. When a strong and weak locator disagree, a dummy invocation is made, thereby updating the locators. This immediately solves the problem if objects cannot migrate, at a cost in additional messages. The dummy invocation can be performed either immediately upon receiving a reference, or after a time-out, or by a bakcground dæmon.

4 Analysis

A proof of the safety of the algorithm, and a discussion of the circumstances under which it exhibits liveness, are in preparation. Essentially, it is shown that the algorithm can never collect non-garbage objects since the scions form a superset of the existing stubs and act as local roots for the LGC. The timestamp windows, as represented by the local threshold vectors, are fundamental to the proof, given the possibility of messages being in-transit or duplicated. Furthermore, for example, if disconnections do not occur after some initial interval, it is shown that all acyclic garbage is eventually collected by the algorithm (note that this is stronger than claiming that the algorithm is live in the absence of disconnections).

Our references require no foreground messages other than the ones sent by the application. Local processing and memory costs appear acceptable. In addition to tolerating non-byzantine failures, the protocols described do not require any form of global synchronization or snapshot, nor are third parties depended upon in any way. As all of the costs incurred are associated with the handling of references, and the background activities need only commence once a reference is used, no overhead is imposed on applications which choose not to use our mechanism.

4.1 Failures

The failure model used in this work is slightly richer than in most comparable material. Messages may be duplicated as well as lost or delivered out-of-order. Processor pairs are subject to periods during which communication between them may be impossible; however, it is not assumed that this failure is either symmetric or transitive. Processors are fail-stop. It is assumed that messages are not undetectably corrupted and that each timestamp generator produces increasing values.

A particular emphasis has been placed on message-related failures in this presentation, as they are most naturally and coherently integrated into our protocols. The support for recovery when some space terminates is more complex and was presented separately. Overall, these mechanisms permit the collection of acyclic garbage in an environment that is rather more demanding than those postulated by most other approaches.

4.2 Costs

The costs of the protocol are considered according to three different measures: in terms of messages, CPU time and memory space. To provide a baseline against which comparisons can be made, consider the state-of-the art implementation, based on UIDs or capabilities, supporting network transparency, but not garbage collection. This system would support messages and timestamps. Data structures would require marshalling and unmarshalling.

This analysis omits the cost of recovery after a crash.

4.2.1 Messages

An important feature of this algorithm is that it requires no additional foreground messages. Additional messages do, however, arise in the background as a consequence of the cleanup protocol.

The marshalled form of a reference, as held in messages, consists of a locator. If stock hardware is used, a marshalled reference is therefore around 16 bytes long. This is comparable to the size of UIDs in many systems. In both our system and the minimal system, messages are timestamped.

Since a UID is location-independent, locating its target entails a distributed search algorithm. In the worse case, a reliable global search is needed. Maintaining a location cache for recently-used UIDs allows to amortize the cost of the search. There is no such cost with locators.

4.2.2 Local CPU Time

Reference marshalling and unmarshalling require searches for existing scions and stubs, prior to creating new ones. A similar cost arises when short-circuiting indirect references. Passing a UID is typically much simpler, involving a simple copy into the message.

A UID system bears a cost searching through its cache for the location of a message's destination. The processing involved is somewhat simpler than our marshalling and the cost is borne only once per message.

Finally, we use additional CPU time in executing the local garbage collector, and in interpreting the live messages of the cleanup protocol. We perform these ac-

tivities in the background, however, so the impact on application performance is minimal.

All the costs listed above are local. A space never needs to wait for another any more than required by the mutator.

4.2.3 Memory

The per-space memory costs of this approach depend on the degree of locality exhibited by the applications executed in the system. Three major data structure types are required: the threshold vector, stubs and scions.

The following estimates of memory usage can be made. Assuming for simplicity of analysis that timestamps, space-identifiers, scion names and local pointers each occupy four bytes, and that all indirection chains and garbage have been eliminated:

- The threshold vector requires one entry, of 8 bytes, for each known remote space.
- There is a single stub for each remote object that is locally referenced, requiring 24 bytes.
- There is a single scion per object for each remote space that contains references to that object; it occupies 20 bytes.

In addition, hash tables or the like are required to implement the accesses modes listed in Section 3.1. These costs are not unreasonable: a maximum of 8+24+20 = 52 bytes per remote reference, across the system as a whole, compares unfavourably, but not appallingly, with the cost of 16-byte UIDs supported by a location cache. Some hotspots may arise, however, if particular well-known objects are referenced from a great many remote spaces, due to the accumulation of scions.

4.3 Measured Performance

We have prototyped an earlier version of our protocol, called SGP [25], on the distributed Lisp Transpive [18]. This version lacks weak locators, uses a message protocol rather than call-reply and uses an extra timestamp vector instead of timestamping stubs. A detailed account and analysis of this experiment may be found in [19].

For our evaluation, we replaced Piquer's original distributed Indirect Reference Count (IRC) collector. Our protocol provides the same functionality as IRC, and is furthermore scalable and resilient to message and space failures.

In this section, we compare the measured performance of our prototype with IRC, in terms of communication and CPU overhead. Our measurements of two applications (merge sort and matrix multiplication) were taken on a Parsytec board composed of four T800 Transputers with one megabyte of memory each, hosted in a Sun. Each application is timed twice in a row; the figures are better the second time because of Transpive's caching policy. The measurements, repeated dozens of times, have shown extremely low variance. Our experiments were able to test resilience to message loss but not to termination, due to lack of a fault-tolerant application. Furthermore, we were not able to quantify how conservative or how scalable our protocol is.

Table 5: Message Overhead

Application	Control Messages					
	IRC		SGP		IRC – SGP	
(sort 100)	31	28	10	8	21	20
(sort 200)	41	39	10	8	31	31
(mult 20 20)	101	96	20	18	81	78

Table 4 shows local execution times. The overhead is due to management of (the Transpive equivalent of) stubs and scions. Our implementation is on average 10% slower than IRC and 20% slower than with distributed collection turned off. This result is encouraging: our implementation is not optimized and retained some obsolete data structures and processing from Piquer's implementation. Furthermore our protocol does more than IRC.

Table 5 measures message overhead. IRC sends "delete" messages, whereas we periodically send live messages. This buffering reduces dramatically the number of control messages.

Although our object model does not take replication into account, it was necessary for Transpive; it proved quite easy to add. But Lisp's extremely fine granularity of objects is very demanding, requiring a huge number of stub and scions which consume a lot of space, increasing the garbage collection overhead.

5 Related Work

This section compares our proposal with related work, in the two areas of location-independent references and distributed garbage collection.

5.1 Location-Independent References

Many distributed systems [17, 21, 24] rely on fixed-length, location-independent Universal IDentifiers (UIDs) to designate and locate objects throughout the network. UIDs do not scale well. Uniqueness can be guaranteed only within some domain; cross-domain references require a separate mechanism. A UID does not carry location information; locating its target entails a global search in the general case. Furthermore, UIDs are not pointers, forcing programmers to use two very different mechanisms.

Table 4: Execution Times

Application	CPU time in seconds						Overhead	
	No DGC		IRC		SGP		SGP/IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5	4.1	17%	5%
(sort 200)	5.6	4.4	6.7	5.2	8.1	5.9	20%	12%
(mult 20 20)	11.1	7.8	12.1	8.7	13.5	9.8	11.9%	12.3%

Our reference mechanism owes much to the links of Demos/MP [20] and the forwarders of Emerald and Hermes [3, 4]. In contrast to their proposals, our references are intimately associated with GC. This is made possible by the invariants maintained by our protocol.

Fowler [10] proposes chaining forwarders to provide continuous access to highly mobile objects. Fowler analyzes three alternative location protocols (distinguished in how they short-circuit indirection chains), demonstrating that the cost decreases dramatically when the number of accesses increases faster than the number of moves. His Jacc protocol bears some similarities with ours. Our stubs carry more information than Fowler's forwarders; for example, our weak locator accesses a non mobile object in a single hop, whereas a forwarder is, in effect, a strong locator. In Fowler's design, a highly mobile object may inform others of its current location, requiring something similar to the source space information in our scions.

5.2 Distributed Garbage Collection

One important problem of distributed garbage collection is maintaining the consistency of scions with stubs in the face of failures. A common approach is to use reliable mechanisms to enforce strong consistency, which is expensive. A more recent approach is to relax traditional GC invariants.

Our scheme is based on the latter alternative and bears similarities to some proposals based on reference counting [6, 18]. Unlike those approaches, however, a scion is maintained per source space, which permits us to tolerate message loss while avoiding the dangers of duplicated delete messages.

Dickman [6] proposes an optimised weighted reference counting (oWRC) algorithm. In order to deal with unreliable communication protocols, oWRC preserves a weak invariant enforcing that each object weight (total weight) is always greater or equal to the sum of all remote reference weights (partial weight). The use of a weak invariant allows the algorithm to tolerate message loss but duplicated messages remain problematic.

Mancini and Shrivastava [15] present an efficient and fault-tolerant reference-counting distributed garbage collector. A reliable RPC mechanism, extended to detect and kill orphans, provides resilience to failures. A special protocol copes with duplication of remote references, by making an early short-cut of potential indirections even if they are never used.

In the future we expect to add to our protocol a separate mechanism to deal with distributed cycles of garbage, which are not currently handled. There are several proposals in the literature, e.g. Bishop's migration technique [2] or Schelvis' cycle-detection technique [22]. We will discuss below Liskov's logically centralized algorithm, Hughes' timestamp algorithm, and Lang's dynamic grouping technique.

Lang *et al.* [13] propose to combine a distributed reference count with the dynamic grouping of nodes with distributed mark-and-sweep within each group. The reference counts must be accurate, hence message failures are not tolerated. The mark-and-sweep algorithm relies heavily on termination protocols, which are not scalable. A distributed garbage cycle that crosses group boundaries is not collected until another group is formed, enclosing the whole cycle; therefore liveness is not guaranteed. A failure during a collection causes group reorganisation excluding the failed node, restarting the group GC.

Hughes [11] uses a global clock. A collector, starting from some local root at time t, marks all objects it reaches with the value t. The marking on a reachable object will advance periodically; on an unreachable object the mark will not change. Objects marked with a date less than some global minimum are collected. Determining the minimum requires repeated execution of a global termination algorithm. Furthermore, if even a single processor is disconnected, it is impossible to advance the minimum.

Liskov and Ladin [14] describe a fault tolerant distributed garbage detector based on their highly available logically-centralised service. Each local collector informs the centralised service of incoming and outgoing references, and about the paths between incoming and outgoing references. The path computation is expensive but necessary for reclamation of distributed garbage cycles. Based on the paths transmitted, the centralised service builds the graph of inter-site references, and detects garbage (includin dead cycles) with a standard tracing algorithm. The centralised service informs LGCs of accessibility of objects.

In a later paper [12] Ladin and Liskov simplify, and correct the deficiencies of, the above proposal, adopting Hughes' algorithm and loosely synchronised local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for for an accu-

rate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralized service determines the garbage threshold date, making a termination protocol unnecessary.

6 Conclusion

We have presented scalable location-transparent references to objects in a distributed system, with well-defined failure semantics. Integrated into the approach is fault-tolerant automatically collection of acyclic distributed garbage, which can be combined with any reasonable local garbage collection algorithm. The mechanism is effective, inexpensive, straightforward and is based on a novel combination of well-known techniques. Our mechanism requires no global search or synchronization and uses a very cheap transport protocol (not requiring multicast communications or any particular ordering on messages). The key enabling concepts are the scions, i.e. inverse reference lists (as opposed to reference counts), and the use of timestamps and windowing protocols to support idempotent deletion. In conjunction a the conservative creation policy, these provide a fault-tolerant and efficient mechanism.

The current specification suffers from some limitations. First, only acyclic garbage is collected; it will be necessary to extend the mechanisms to collect distributed cyclic garbage. Second, recovery from space termination is incompletely specified. Third, although the main-line protocol is scalable, the recovery protocol entails global search; to limit the cost of search, we pointed at the need to structure the universe into small partitions (in which exhaustive search remains realistic) connected by gateways, but this needs more work.

A first version of the garbage collection protocol has been prototyped; its measured performance is similar to an existing, non fault-tolerant, non scalable, distributed collector. We are currently in the process of implementing the specifications of this paper, as a system level facility in the Soul object-support layer [23].

Acknowledgments

We particularly wish to thank Daniel Edelson for his helpful comments and discussions, especially those concerning the presentation of this material. We would also like to thank Olivier Gruber, Bernard Lang and Robert Cooper for their comments on earlier versions of this work.

References

[1] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Western Research Laboratory, Palo Alto, CA (USA), February 1988.

[2] P. B. Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Institute of Technology, MIT Laboratory for Computer Science, Cambridge MA (USA), May 1977.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 1986.

[4] Andrew P. Black and Yeshayahu Artsy. Implementing location independant invocation. In *Proceedings of the 9th Int. Conf. on Distributed Computing Systems*, pages 550–559, Newport Beach, CA USA, June 1989. IEEE.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[6] Peter Dickman. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems. Submitted for publication, 1992.

[7] Peter W. Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, University of Cambridge Computer Laboratory, 1992.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[9] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Principles of Programming Languages*, pages 51–57, Albuquerque, NM (USA), January 1992.

[10] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. 5th Annual ACM Symp. on Principles of Distributed Computing*, pages 108–120, Alberta, Canada, August 1986.

[11] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[12] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.*, Yokohama (Japan), 1992.

[13] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

[14] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[15] L. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Jounal*, 34(6):503–513, December 1991.

[16] William Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, 1980.

[17] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[18] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume I of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[19] David Plainfossé and Marc Shapiro. Experience with a fault-tolerant garbage collector in a distributed lisp system. Submitted for publication, April 1992.

[20] M.L. Powell and B.P Miller. Process migration in Demos/MP. In *9th ACM Symposium on Operating System Principles*, volume 17, pages 110–119, October 1983.

[21] Marc Rozier, Vadim Abrossimov, Franccois Armand, Ivan Boule, Frédéric Herrmann, Michel Gien, Marc Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, December 1988.

[22] Marcel Schelvis. Incremental distribution of timestamp packets: a new approach to distributed garbage collection. In Norman Meyrowitz, editor, *OOPSLA'89 Conf. Proc.*, volume 24 of *SIGPLAN Notices*, pages 37–48, New Orleans, LA (USA), October 1989. ACM Sigplan, ACM.

[23] Marc Shapiro. Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*, pages 251–255, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[24] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[25] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

The Weakest Failure Detector for Solving Consensus*

Tushar Deepak Chandra[†§] Vassos Hadzilacos[‡] Sam Toueg[§]

Abstract

We determine what information about failures is necessary and sufficient to solve Consensus in asynchronous distributed systems subject to crash failures. In [CT91], we proved that $\diamond\mathcal{W}$, a failure detector that provides surprisingly little information about which processes have crashed, is sufficient to solve Consensus in asynchronous systems with a majority of correct processes. In this paper, we prove that to solve Consensus, any failure detector has to provide at least as much information as $\diamond\mathcal{W}$. Thus, $\diamond\mathcal{W}$ is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes.

1 Introduction

1.1 Background

The asynchronous model of distributed computing has been extensively studied. Informally, an *asynchronous distributed system* is one in which message transmission times and relative processor speeds are both unbounded. Thus an algorithm designed for an asynchronous system does not rely on such bounds for its correctness. In practice, asynchrony is introduced by unpredictable loads on the system.

Although the asynchronous model of computation is attractive for the reasons outlined above, it is well-known that many fundamental problems of fault-tolerant distributed computing that are solvable in synchronous systems, are unsolvable in asynchronous systems. In particular, it is well-known that *Consensus*, and several forms of reliable broadcast, including *Atomic Broadcast*, cannot be solved deterministically in an asynchronous system that is subject to even a single *crash* failure [FLP85, DDS87]. Essentially, these impossibility results stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow".

To circumvent these impossibility results, previous research focused on the use of randomization techniques [CD89], the definition of some weaker problems and their solutions [DLP+86, ABND+87, BW87], or the study of several models of *partial synchrony* [DDS87, DLS88]. However, the impossibility of deterministic solutions to many *agreement* problems (such as Consensus and Atomic Broadcast) remains a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed computing.

An alternative approach to circumvent such impossibility results is to augment the asynchronous model of computation with a *failure detector*. Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about which processes may have crashed so far: Each process

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp, and a grant from the Natural Sciences and Engineering Research Council of Canada.

†Also supported by an IBM graduate fellowship.

‡Computer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 1A1

§Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853

PoDC '92-8/92/B.C.

• 1992 ACM 0-89791-496-1/92/0008/0147...$1.50

has access to a local *failure detector module* that monitors other processes in the system, and maintains a list of those that it currently suspects to have crashed. Each process periodically consults its failure detector module, and uses the list of suspects returned in solving Consensus.

A failure detector module can make *mistakes* by erroneously adding processes to its list of suspects: i.e., it can suspect that a process p has crashed even though p is still running. If it later believes that suspecting p was a mistake, it can remove p from its list. Thus, each module may repeatedly *add* and *remove* processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects.

It is important to note that the mistakes made by a failure detector should not prevent any correct process from behaving according to specification. For example, consider an algorithm that uses a failure detector to solve Atomic Broadcast in an asynchronous system. Suppose all the failure detector modules wrongly (and permanently) suspect that a correct process p has crashed. The Atomic Broadcast algorithm must still ensure that p delivers the same set of messages, in the same order, as all the other correct processes. Furthermore, if p broadcasts a message m, all correct processes must deliver m.[1]

In [CT91], we showed that a surprisingly weak failure detector is sufficient to solve Consensus and Atomic Broadcast in asynchronous systems with a majority of correct processes. This failure detector, called the *eventually weak failure detector* and denoted $\mathcal{W}$ here, satisfies only the following two properties:[2]

1. There is a time after which every process that crashes is always suspected by some correct process.

2. There is a time after which some correct process is never suspected by any correct process.

Note that, at any given time t, processes cannot use $\mathcal{W}$ to determine the identity of a correct process. Furthermore, they cannot determine whether there is a correct process that will not be suspected after time t.

The failure detector $\mathcal{W}$ can make an *infinite* number of mistakes. In fact, it can forever add and then remove some *correct* processes from the lists of suspects (this reflects the inherent difficulty of determining whether a process is just slow or has crashed). Moreover, some correct processes may be erroneously suspected to have crashed by all the other processes throughout the entire execution.

The two properties of $\mathcal{W}$ state that eventually something must hold forever; this may appear too strong a requirement to implement in practice. However, when solving a problem that "terminates", such as Consensus, it is not really required that the properties hold *forever*, but merely that they hold for a *sufficiently long time*, i.e., long enough for the algorithm that uses the failure detector to achieve its goal. For instance, in practice the algorithm of [CT91] that solves Consensus using $\mathcal{W}$ only needs the two properties of $\mathcal{W}$ to hold for a relatively short period of time.[3] However, in an asynchronous system it is not possible to quantify "sufficiently long", since even a single process step or a single message transmission is allowed to take an arbitrarily long amount of time. Thus it is convenient to state the properties of $\mathcal{W}$ in the stronger form given above.

1.2 The problem

The failure detection properties of $\mathcal{W}$ are *sufficient* to solve Consensus in asynchronous systems. But are they *necessary*? For example, consider failure detector $\mathcal{A}$ that satisfies Property 1 of $\mathcal{W}$ and the following weakening of Property 2:

> There is a time after which some correct process is never suspected by at least 99% of the correct processes.

[1] A different approach was taken in [RB91]: a correct process that is wrongly suspected to have crashed, voluntarily leaves the system. It may later rejoin the system by assuming a new identity.

[2] In [CT91], this was denoted $\diamond\mathcal{W}$.

[3] In that algorithm processes are cyclically elected as "coordinators". Consensus is achieved as soon as a correct coordinator is reached, and no process suspects it to have crashed while this coordinator is trying to enforce consensus.

$\mathcal{A}$ is clearly weaker than $\mathcal{W}$. Is it possible to solve Consensus using $\mathcal{A}$? Indeed what is the *weakest* failure detector *sufficient* to solve Consensus in asynchronous systems? In trying to answer this fundamental question we run into a problem. Consider failure detector $\mathcal{B}$ that satisfies the following two properties:

1. There is a time after which every process that crashes is always suspected by *all* correct processes.

2. There is a time after which some correct process is never suspected by a majority of the processes.

It seems that $\mathcal{B}$ and $\mathcal{W}$ are *incomparable*: $\mathcal{B}$'s first property is stronger than $\mathcal{W}$'s, and $\mathcal{B}$'s second property is weaker than $\mathcal{W}$'s. Is it possible to solve Consensus in an asynchronous system using $\mathcal{B}$? The answer turns out to be "yes" (provided that this asynchronous system has a majority of correct processes, as $\mathcal{W}$ also requires). Since $\mathcal{W}$ and $\mathcal{B}$ appear to be incomparable, one may be tempted to conclude that $\mathcal{W}$ cannot be the "weakest" failure detector with which Consensus is solvable. Even worse, it raises the possibility that no such "weakest" failure detector exists.

However, a closer examination reveals that $\mathcal{B}$ and $\mathcal{W}$ are indeed comparable in a natural way: There is a distributed algorithm $T_{\mathcal{B}\rightarrow\mathcal{W}}$ that can transform $\mathcal{B}$ into a failure detector with the Properties 1 and 2 of $\mathcal{W}$. $T_{\mathcal{B}\rightarrow\mathcal{W}}$ works for any asynchronous system that has a majority of correct processes. We say that $\mathcal{W}$ is *reducible to* $\mathcal{B}$ in such a system. Since $T_{\mathcal{B}\rightarrow\mathcal{W}}$ is able to transform $\mathcal{B}$ into $\mathcal{W}$ in an asynchronous system, $\mathcal{B}$ must provide at least as much information about process failures as $\mathcal{W}$ does. Intuitively, $\mathcal{B}$ is at least as strong as $\mathcal{W}$.

1.3 The result

In [CT91], we showed that $\mathcal{W}$ is sufficient to solve Consensus in asynchronous systems if and only if $n > 2f$ (where n is the total number of processes, and f is the maximum number of processes that may crash). In this paper, we prove that $\mathcal{W}$ is reducible to *any* failure detector D that can be used to solve Consensus (this result holds for any asynchronous system). We show this reduction by giving a distributed algorithm $T_{D\rightarrow\mathcal{W}}$ that transforms any such D into $\mathcal{W}$. Therefore, $\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in asynchronous systems with $n > 2f$. Furthermore, if $n \leq 2f$, any failure detector that can be used to solve Consensus must be strictly stronger than $\mathcal{W}$.

The task of transforming any given failure detector D (that can be used to solve Consensus) into $\mathcal{W}$ runs into a serious technical difficulty for the following reasons:

- To strengthen our result, we do not restrict the output of D to lists of suspects. Instead, this output can be *any value* that encodes some information about failures. For example, a failure detector D should be allowed to output any boolean formula, such as "(not p) and (q or r)" (i.e., p is up and either q or r has crashed)—or any *encoding* of such a formula. Indeed, the output of D could be an arbitrarily complex (and unknown) encoding of failure information. Our transformation from D into $\mathcal{W}$ must be able to decode this information.

- Even if the failure information provided by D is not encoded, it is not clear how to extract from it the failure detection properties of $\mathcal{W}$. Consequently, if D is given in isolation, the task of transforming it into $\mathcal{W}$ may not be possible.

Fortunately, since D can be used to solve Consensus, there is a corresponding algorithm, $Consensus_D$, that is somehow able to "decode" the information about failures provided by D, and knows how to use it to solve Consensus. Our reduction algorithm, $T_{D\rightarrow\mathcal{W}}$ uses $Consensus_D$ to extract this information from D and transforms it into the properties of $\mathcal{W}$.

2 The model

We describe a model of asynchronous computation with failure detection patterned after the one in [FLP85].

2.1 Failure Detectors

We assume the existence of a discrete global clock to simplify the presentation. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

The system consists of a set of n *processes*, $\Pi = \{p_1, p_2, \ldots, p_n\}$ that may fail by crashing. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have *crashed* through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we say p *crashes in* F and if $p \in \mathit{correct}(F)$ we say p *is correct in* F.

Associated with each failure detector is a range $\mathcal{R}$ of values output by that failure detector. A *failure detector history* H *with range* $\mathcal{R}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $H(p,t)$ is the value of the failure detector module of process p at time t. A *failure detector* D is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_D$ (where $\mathcal{R}_D$ denotes the range of failure detector outputs of D). $D(F)$ denotes the set of possible failure detector histories permitted by D for the failure pattern F.

For example, consider the failure detector $\mathcal{W}$ mentioned in the introduction. Each failure detector module of $\mathcal{W}$ outputs a *set of processes* that are suspected to have crashed: in this case $\mathcal{R}_{\mathcal{W}} = 2^{\Pi}$. For each failure pattern F, $\mathcal{W}(F)$ is the set of all failure detector histories $H_{\mathcal{W}}$ with range $\mathcal{R}_{\mathcal{W}}$ that satisfy the following properties:

1. There is a time after which every process that crashes in F is always suspected by some process that is correct in F:

$$\exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \exists q \in \mathit{correct}(F), \forall t' \geq t : p \in H_{\mathcal{W}}(q, t')$$

2. There is a time after which some process that is correct in F is never suspected by any process that is correct in F:

$$\exists t \in \mathcal{T}, \exists p \in \mathit{correct}(F), \forall q \in \mathit{correct}(F), \forall t' \geq t : p \notin H_{\mathcal{W}}(q, t')$$

Note that we *specify* a failure detector D as a function of the failure pattern F of an execution. However, this does *not* preclude an *implementation* of D from using other aspects of the execution such as when messages are received. Thus, executions with the same failure pattern F may still have different failure detector histories. It is for this reason that we allow $D(F)$ to be a *set* of failure detector histories from which the actual failure detector history for a particular execution is selected non-deterministically.

2.2 Algorithms

We model the asynchronous communication channels as a *message buffer* which contains messages of the form (p, data, q) indicating that process p has sent *data* addressed to process q and q has not yet received that message. An *algorithm* A is a collection of n (possibly infinite state) deterministic automata, one for each of the processes. $A(p)$ denotes the automaton running on process p. Computation proceeds in *steps of the given algorithm* A. In each step of A, process p performs atomically the following three phases:

Receive phase: p receives a single message of the form (q, data, p) from the message buffer, or a "null" message, denoted λ, meaning that no message is received by p during this step.

Failure detector query phase: p queries and receives a value from its failure detector module. We say that p *sees a value* d when the value returned by p's failure detector module is d.

Send phase: p changes its state and sends a message to all the processes according to the automaton $A(p)$, based on its state at the beginning of the step, the message received in the receive phase, and the value that p sees in the failure detector query phase.[4]

[4] In the send phase, p sends a message to all the processes atomically. As was shown in [FLP85], the ability to do so is *not* sufficient for solving Consensus. An alternative formulation of a step could restrict a process to sending a message to a single process in the send phase. We can show that both formulations are equivalent for our purposes.

The message actually received by the process p in the receive phase is chosen *non-deterministically* from amongst the messages in the message buffer destined to p, and the null message λ. The null message may be received *even if* there are messages in the message buffer that are destined to p: the fact that m is in the message buffer merely indicates that m was sent to p. Since ours will be a model of asynchronous systems, where messages may experience arbitrary (but finite) delays, the amount of time m may remain in the message buffer before it is received is unbounded. Though message delays are arbitrary, we also want them to be finite. We model this by introducing a liveness assumption: every message sent will eventually be received, provided its recipient makes "sufficiently many" attempts to receive messages. All this will be made more precise later.

To keep things simple we assume that a process p sends a message m to q at most once. This allows us to speak of the contents of the message buffer as a set, rather than a multiset. We can easily enforce this by adding a counter to each message sent by p to q — so this assumption does not damage generality.

2.3 Configurations, Runs and Environments

A *configuration* is a pair (s, M), where s is a function mapping each process p to its local state, and M is a set of triples of the form $(q, data, p)$ representing the messages presently in the message buffer. An *initial configuration of an algorithm A* is a configuration (s, M), where $s(p)$ is an initial state of $A(p)$ and $M = \emptyset$. A *step* of a given algorithm A transforms one configuration to another. A step of A is uniquely determined by the identity of the process p that takes the step, the message m received by p during that step, and the failure detector value d seen by p during the step. Thus, we identify a step of A with a tuple (p, m, d, A) ($m = \lambda$ when the null message is received). We say that a step $e = (p, m, d, A)$ *is applicable to a configuration* $C = (s, M)$ if and only if $m \in M \cup \{\lambda\}$. We write $e(C)$ to denote the unique configuration that results when e is applied to C.

A *schedule S of algorithm A* is a (possibly finite) sequence of steps of A. $S_\perp$ denotes the empty schedule. We say that a schedule S of an algorithm A *is applicable to a configuration C* if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc.[5] If S is a finite schedule applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C. Note $S_\perp(C) = C$ for all configurations C.

A *partial run of algorithm A using a failure detector D* is a tuple $R = \langle F, H_D, I, S, T \rangle$ where F is a failure pattern, $H_D \in D(F)$ is a failure detector history, I is an initial configuration of A, S is a *finite* schedule of A, and T is a *finite* list of increasing time values (indicating when each step in S occurred) such that $|S| = |T|$, S is applicable to I, and for all $i \leq |S|$, if $S[i]$ is of the form (p, m, d, A) then:

- p has not crashed by time $T[i]$, i.e., $p \notin F(T[i])$
- d is the value of the failure detector module of p at time $T[i]$, i.e., $d = H_D(p, T[i])$

Informally, a partial run of A using D represents a finite point of some execution of A using D.

A *run of an algorithm A using a failure detector D* is a tuple $R = \langle F, H_D, I, S, T \rangle$ where F is a failure pattern, $H_D \in D(F)$ is a failure detector history, I is an initial configuration of A, S is an *infinite* schedule of A, and T is an *infinite* list of increasing time values indicating when each step in S occurred. In addition to satisfying the above properties of a partial run, a run must also satisfy the following properties:

- Every correct process takes an infinite number of steps in S.
- Every message sent to a correct process is eventually received.

In [CT91], we proved that any algorithm that uses $\mathcal{W}$ to solve Consensus requires $n > 2f$. With other failure detectors the requirements may be different. For example, there is a failure detector that can be used to solve Consensus only if p_1 and p_2 do not both crash. In general whether a given

[5] We denote by $v[i]$ the ith element of a sequence v.

failure detector can be used to solve Consensus depends upon assumptions about the underlying "environment". Formally, an *environment* $\mathcal{E}$ (of an asynchronous system) is set of possible failure patterns.

3 The Consensus problem

In the Consensus problem, each process p has an initial value, 0 or 1, and must reach an irrevocable decision on one of these values.

We say that algorithm A *uses failure detector* D *to solve Consensus in environment* $\mathcal{E}$ if every run $R = \langle F, H_D, I, S, T \rangle$ of A using D where $F \in \mathcal{E}$ satisfies:

Termination: Each correct process eventually decides.

Validity: Each correct process decides on the initial value of some process.

Agreement: No two correct processes decide differently.

4 Reducibility

We now define what it means for an algorithm $T_{D \to D'}$ to transform a failure detector D into another failure detector D' in an environment $\mathcal{E}$. Algorithm $T_{D \to D'}$ uses D to maintain a variable $output_p$ at every process p. This variable, reflected in the local state of p, emulates the output of D' at p. Let O_R be the history of all the *output* variables in run R, i.e., $O_R(p,t)$ is the value of $output_p$ at time t in run R. Algorithm $T_{D \to D'}$ *transforms* D *into* D' *in* $\mathcal{E}$ if and only if for every run $R = \langle F, H_D, I, S, T \rangle$ of $T_{D \to D'}$ using D, where $F \in \mathcal{E}$, $O_R \in D'(F)$.

Given $T_{D \to D'}$, anything that can be done using D' in $\mathcal{E}$, can be done using D instead. To see this, suppose a given algorithm B requires failure detector D' (when it executes in $\mathcal{E}$), but only D is available. We can still execute B as follows. Concurrently with B, we run $T_{D \to D'}$ to transform D into D'. We now modify the failure detector query phase of each step of B at process p: p reads the current value of $output_p$ (which is concurrently maintained by $T_{D \to D'}$) instead of querying its failure detector module. This is illustrated in Fig. 1.

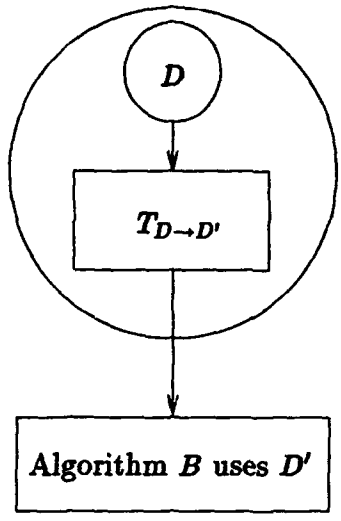


Figure 1: Transforming D into D'

Intuitively, since $T_{D \to D'}$ is able to use D to emulate D', D provides at least as much information about process failures in $\mathcal{E}$ as D' does. Thus, if there is an algorithm $T_{D \to D'}$ that transforms D into D' in $\mathcal{E}$, we write $D \succeq_{\mathcal{E}} D'$ and say that D' *is reducible to* D *in* $\mathcal{E}$; we also say that D' *is weaker than* D *in* $\mathcal{E}$.

5 An outline of the result

In [CT91] we showed that $\mathcal{W}$ can be used to solve Consensus in any environment in which $n > 2f$. We now show that $\mathcal{W}$ is weaker than any failure detector that can be used to solve Consensus. This result holds for any environment $\mathcal{E}$. Together with [CT91], this implies that $\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in any environment in which $n > 2f$.

To prove our result, we first define a new failure detector, denoted Ω, that is at least as strong as $\mathcal{W}$. We then show that any failure detector D that can be used to solve Consensus is at least as strong as Ω. Thus, D is at least as strong as $\mathcal{W}$.

The output of the failure detector module of Ω at a process p is a *single* process, q, that p currently

considers to be *correct*; we say that p *trusts* q. In this case, $\mathcal{R}_\Omega = \Pi$. For each failure pattern F, $\Omega(F)$ is the set of all failure detector histories H_Ω with range $\mathcal{R}_\Omega$ that satisfy the following property:

- There is a time after which all the correct processes always trust the same correct process:

$$\exists t \in \mathcal{T}, \exists q \in \mathit{correct}(F), \forall p \in \mathit{correct}(F), \\ \forall t' \geq t : H_\Omega(p, t') = q$$

As with $\mathcal{W}$, the output of the failure detector module of Ω at a process p may change with time, i.e., p may trust different processes at different times. Furthermore, at any given time t, processes p and q may trust different processes.

Theorem 1: For all environments $\mathcal{E}$, $\Omega \succeq_\mathcal{E} \mathcal{W}$.
Proof: [Sketch] The reduction algorithm $T_{\Omega \to \mathcal{W}}$ that transforms Ω into $\mathcal{W}$ is as follows. Each process p, periodically sets $\mathit{output}_p \leftarrow \Pi - \{q\}$, where q is the process that p currently trusts according to Ω. It is easy to see that (in any environment $\mathcal{E}$) this output satisfies the two properties of $\mathcal{W}$. □

Theorem 2: For all environments $\mathcal{E}$, if a failure detector D can be used to solve Consensus in $\mathcal{E}$, then $D \succeq_\mathcal{E} \Omega$.
Proof: The reduction algorithm $T_{D \to \Omega}$ is shown in Section 6. It is the core of our result. □

Corollary 3: For all environments $\mathcal{E}$, if a failure detector D can be used to solve Consensus in $\mathcal{E}$, then $D \succeq_\mathcal{E} \mathcal{W}$.

In [CT91] we proved that, for all environments $\mathcal{E}$ in which $n > 2f$, $\mathcal{W}$ can be used to solve Consensus. Together with Corollary 3, this shows that:

Corollary 4: For all environments $\mathcal{E}$ in which $n > 2f$, $\mathcal{W}$ is the weakest failure detector that can be used to solve Consensus in $\mathcal{E}$.

6 The reduction algorithm

Let $\mathcal{E}$ be an environment, D be a failure detector that can be used to solve Consensus in $\mathcal{E}$, and $\mathit{Consensus}_D$ be the Consensus algorithm that uses D. We describe an algorithm $T_{D \to \Omega}$ that transforms D into Ω in $\mathcal{E}$. Intuitively, this algorithm works as follows. Fix an arbitrary run of $T_{D \to \Omega}$ using D in $\mathcal{E}$, with failure pattern $F \in \mathcal{E}$, and failure detector history $H_D \in D(F)$. We shall first construct an infinite directed acyclic graph, denoted G, whose vertices are some of the failure detector values that occur in H_D, and whose edges are consistent with the time at which these values occur. We then show that G induces a simulation forest Υ that encodes an infinite set of possible runs of $\mathit{Consensus}_D$. Finally, we show how to extract from Υ the identity of a process p^* that is correct in F.

The induced simulation forest is infinite and thus it cannot be computed by any process. However, the information needed to extract p^* is present in a *finite* subgraph of the forest. It will be sufficient for each correct process p to construct ever increasing finite approximations of the simulation forest Υ that will eventually include this crucial finite subgraph. At all times, p uses its present approximation of Υ to select the identity of some process: once p's approximation of Υ includes the crucial finite subgraph, the selected process will be p^* (forever). Thus, there is a time after which all correct processes trust the same correct process, p^*—which is exactly what Ω requires.

We say that a process is *correct* (*crashes*) if it is correct (crashes) in F. For simplicity, we assume that a process p sees a value d at most once (this can be enforced by tagging a counter to each value seen). For the rest of this paper, whenever we refer to a run of $\mathit{Consensus}_D$, we mean a run of $\mathit{Consensus}_D$ using D. Furthermore, we only consider schedules of $\mathit{Consensus}_D$, and therefore we write (p, m, d) instead of $(p, m, d, \mathit{Consensus}_D)$ to denote a step.

6.1 A DAG and a forest

Given the failure pattern F and the corresponding failure detector history $H_D \in D(F)$ that were fixed above, let G be any infinite directed acyclic graph with the following properties:

1. The vertices of G are of the form $[p, d]$ where $d = H_D(p, t)$ for some time t.

2. If $[q_1, d_1] \rightarrow [q_2, d_2]$ is an edge of G and $d_1 = H_D(q_1, t_1)$ and $d_2 = H_D(q_2, t_2)$ then $t_1 < t_2$.

3. G is transitively closed.

4. Let p be any correct process and V be a finite subset of vertices of G. There is a failure detector value d such that for all vertices $[p', d']$ in V, $[p', d'] \rightarrow [p, d]$ is an edge of G.

Note that such a DAG represents only a "sampling" of the failure detector values that occur in H_D. In particular, we do not require that it contain all the values that occur in H_D or that it relate (with an edge) all the values according to the time at which they occur.

Let $g = [q_1, d_1], [q_2, d_2], \ldots$ be any (finite or infinite) path of G. A schedule S is *compatible with* g if it has the same length as g, and $S = (q_1, m_1, d_1), (q_2, m_2, d_2), \ldots$, for some (possibly null) messages $m_1, m_2, \ldots$; S is *compatible with* G if it is compatible with some path of G. S is *induced by* g *and an initial configuration* I (of $Consensus_D$) if S is compatible with g and applicable to I. S is *induced by* G *and* I if S is compatible with G and applicable to I. Note that each g and I induce several schedules, each corresponding to a different sequence of messages received.

Lemma 5: Let S be any finite schedule induced by G and some initial configuration I of $Consensus_D$. There is a T such that $\langle F, H_D, I, S, T \rangle$ is a partial run of $Consensus_D$.

Lemma 6: Let S be any infinite schedule induced by G and some I, such that every correct process takes an infinite number of steps and every message sent to a correct process is eventually received. There is a T such that $\langle F, H_D, I, S, T \rangle$ is a run of $Consensus_D$.

The set of schedules that are induced by G and some particular I, can be organized as a tree, the *simulation tree* Υ_G^I *induced by* G *and* I. These schedules are the vertices of the tree, with (the empty schedule) $S_\perp$ at the root. There is an edge from S to S' if and only if $S' = S \cdot e$ for a step e.

Lemma 7: Let S be any vertex of Υ_G^I and p be any correct process. Let m be a message in the message buffer of $S(I)$ addressed to p or the null message. Υ_G^I has a vertex $S \cdot (p, m, d)$ for some d.

Lemma 8: Let $S, S_1, S_2, \ldots, S_k$ be vertices of Υ_G^I. There is a schedule E containing only steps of correct processes such that:

1. $S \cdot E$ is a vertex of Υ_G^I and all correct processes have decided in $S \cdot E(I)$.

2. $S_i \cdot E$ $(1 \leq i \leq k)$ is compatible with G.

Note that E may not be applicable to $S_i(I)$, and thus $S_i \cdot E$ is not necessarily a vertex of Υ_G^I.

Let I^i, $0 \leq i \leq n$ denote the initial configuration of $Consensus_D$ in which the initial values of $p_1 \ldots p_i$ are 1, and the initial values of $p_{i+1} \ldots p_n$ are 0. We define the *simulation forest induced by* G to be the set of $n+1$ simulation trees induced by G and these initial configurations.

6.2 Tagging the simulation forest

We assign a set of tags to each vertex of each tree $\Upsilon_G^{I^i}$ in the simulation forest induced by G. Vertex S of $\Upsilon_G^{I^i}$ receives tag k if and only if it has a descendent S' such that some correct process has decided k in $S'(I^i)$. Hereafter, Υ^i denotes the tagged tree $\Upsilon_G^{I^i}$, and Υ denotes the tagged simulation forest.

Lemma 9: Each vertex of Υ^i has at least one tag.

A vertex of Υ^i is *monovalent* if it has only one tag, and *bivalent* if it has both tags, 0 and 1. A vertex is *0-valent* if it is monovalent and is tagged 0; *1-valent* is similarly defined.

Lemma 10: The ancestors of a bivalent vertex are bivalent. The descendents of a k-valent vertex are k-valent.

Lemma 11: If S is a bivalent vertex of Υ^i then no correct process has decided in $S(I^i)$.

Recall that in I^0 all processes have initial value 0, while in I^n they all have initial value 1.

Lemma 12: The root of Υ^0 is 0-valent and the root of Υ^n is 1-valent.

If the root of Υ^i is bivalent, then i is *bivalent critical*. If the root of Υ^{i-1} is 0-valent but the root of Υ^i is 1-valent, then i is *monovalent critical*. Index i is *critical* if it is monovalent or bivalent critical.

Lemma 13: There is a critical i, $0 < i \leq n$.

The critical index i is the key to extracting the identity of a correct process. In fact, if i is monovalent critical, we shall prove that p_i must be correct (Lemma 15). If i is bivalent critical, the correct process will be found by focusing on the tree Υ^i, as explained in the following section.

6.3 Of hooks and forks

We describe two types of finite subtrees of Υ^i referred to as *decision gadgets of* Υ^i. Each type of decision gadget is rooted at $S_\perp$ and has exactly two leaves: one 0-valent and one 1-valent. The least common ancestor of these leaves is called the *pivot*. The pivot is clearly bivalent.

The first type of decision gadget is called a *fork*, and is shown in Figure 2. The two leaves are children of the pivot, obtained by applying different steps of the *same* process p. Process p is the *deciding process of the fork*, because its step after the pivot determines the decision of correct processes.

The second type of decision gadget is called a *hook*, and is shown in Figure 3. Let S be the pivot of the hook. There is a step e such that $S \cdot e$ is one leaf, and the other leaf is $S \cdot (p, m, d) \cdot e$ for some p, m, d. Process p is the *deciding process of the hook*, because the decision of correct process is determined by whether p takes the step (p, m, d) before e.

We shall prove that the deciding process p of a gadget must be correct (Lemma 16). Intuitively, this is because if p crashes no process can figure out whether p has taken the step that determines the decision value. The existence of such a critical "hidden" step is also at the core of many impossibility proofs starting with [FLP85]. In our case, the "hiding" is more difficult because now processes have recourse to the failure detector D. Despite this, the hiding of the step of the deciding process of a gadget is still possible. The key to proving this is Lemma 8.

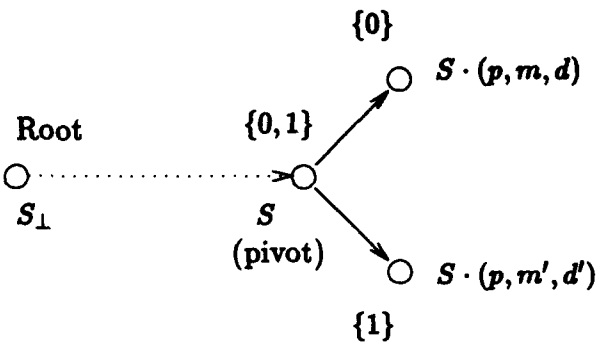


Figure 2: A fork—p is the deciding process

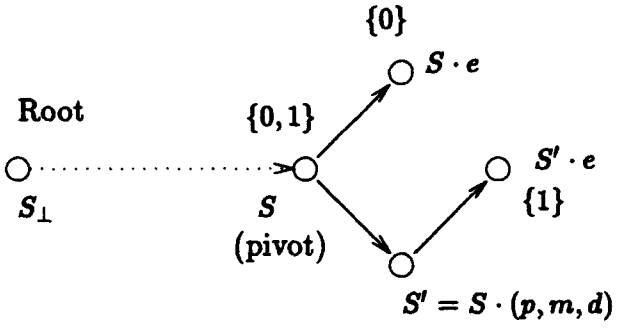


Figure 3: A hook—p is the deciding process

Lemma 14: If i is bivalent critical then Υ^i has at least one decision gadget (and hence a deciding process).

6.4 Extracting the correct process

By Lemma 13, there is a critical index i. If i is monovalent critical, Lemma 15 below shows how to extract a correct process. If i is bivalent critical, a correct process can be found by applying Lemmata 14 and 16.

Lemma 15: If i is monovalent critical then p_i is correct.

Lemma 16: The deciding process of a decision gadget is correct.

There may be several critical indices and several

{*Build and tag simulation forest* Υ *induced by* G}
for $i \leftarrow 0, 1, \ldots, n$:
  $\Upsilon^i \leftarrow$ simulation tree induced by G and I^i
  for every vertex S of Υ^i
    if S has a descendent S' such that
      a correct process has decided k in $S'(I^i)$
    then add tag k to S

{*Select a process from tagged simulation forest* Υ}
$i \leftarrow$ smallest critical index
if i is monovalent critical **then return** p_i
else return deciding process
    of the smallest gadget in Υ^i

Figure 4: Selecting a correct process

decision gadgets in the simulation forest. Thus, the above Lemmata may identify many correct processes. Our selection rule will choose *one* of these, as the failure detector Ω requires, as follows. It first determines the smallest critical index i. If i is monovalent critical, it selects p_i. If, on the other hand, i is bivalent critical, it chooses the "smallest" gadget in Υ^i according to some encoding of gadgets, and selects the corresponding deciding process. It is easy to encode finite graphs as natural numbers. Since a gadget is just a finite graph, the selection rule can use any such encoding. The selection rule is shown in Figure 4.

Lemma 17: Figure 4 selects a correct process.

6.5 The reduction algorithm $T_{D \to \Omega}$

The selection of a correct process described above is not yet the distributed algorithm $T_{D \to \Omega}$ that we are seeking: it involved an infinite simulation forest and it was "centralized". To turn it into a distributed algorithm, we will modify it as follows. Each process will cooperate with other processes to construct ever increasing finite approximations of the simulation forest. Such approximations will eventually contain the gadget and the other tagging information necessary to extract the identity of the *same* correct process chosen by the selection method in Figure 4.

Note that the selection method in Figure 4 involves three stages: The construction of G, a graph representing samples of failure detector values and their temporal relationship, the construction and tagging of the simulation forest induced by G, and finally, the selection of a correct process using this forest.

Algorithm $T_{D \to \Omega}$ consists of two components. In the first component, each process repeatedly queries its failure detector module and sends the failure detector values it sees to the other processes. This component enables processes to construct ever increasing finite approximations of the *same* G. Since all inter-process communication occurs in this component, we call it the *communication component* of $T_{D \to \Omega}$.

In the second component, each process repeatedly (a) constructs and tags the simulation forest induced by its current approximation of G, and (b) selects the identity of a process using its current simulation forest. Since this component does not require any communication, we call it the *computation component* of $T_{D \to \Omega}$.

6.5.1 The communication component

In this component processes cooperate to construct ever increasing approximations of the same G. Let G_p denote p's current approximation of G. Roughly, each process p repeatedly executes: (i) If p reccives G_q for some q, it incorporates this information by replacing G_p with the union of G_p and G_q. (ii) Process p queries its own failure detector module. Let d be the value that it sees and $[p', d']$ be any vertex currently in G_p. Clearly, p saw d after p' saw d'. Thus p adds $[p, d]$ to G_p, with edges from all other vertices of G_p to $[p, d]$. Process p then sends its updated G_p to all other processes. The communication component of $T_{D \to \Omega}$ for p is shown in Figure 5.

Recall that we are considering a fixed run of $T_{D \to \Omega}$, with failure pattern F, and failure detector history $H_D \in D(F)$. The communication component of $T_{D \to \Omega}$ constructs graphs that satisfy the following properties. Let $G_p(t)$ denote the value of G_p at time t.

Lemma 18: For any correct process p and $t \in \mathcal{T}$:

1. The vertices of $G_p(t)$ are of the form $[p', d']$

{Build the directed acyclic graph G_p}
$G_p \leftarrow$ empty graph
repeat forever
  RECEIVE PHASE:
    p receives m
  FAILURE DETECTOR QUERY PHASE:
    $d_p \leftarrow$ query failure detector D
  SEND PHASE:
    if m is of the form (q, G_q, p) **then**
      $G_p \leftarrow G_p \cup G_q$
    add $[p, d_p]$ to G_p and edges from all other vertices of G_p to $[p, d_p]$
    output$_p \leftarrow$ *computation component {Fig. 6}*
    p sends (p, G_p, q) to all $q \in \Pi$

Figure 5: Process p's communication component

where $d' = H_D(p', t')$ for some time t'.

2. If $[q_1, d_1] \rightarrow [q_2, d_2]$ is an edge of $G_p(t)$ and $d_1 = H_D(q_1, t_1)$ and $d_2 = H_D(q_2, t_2)$ then $t_1 < t_2$.

3. $G_p(t)$ is transitively closed.

4. There is a time $t' \geq t$ and a failure detector value d such that for all vertices $[p', d']$ of $G_p(t)$, $[p', d'] \rightarrow [p, d]$ is an edge of $G_p(t')$.

5. $G_p(t)$ is a subgraph of $G_p(t+1)$.

6. For all correct q, there is a time $t' \geq t$ such that $G_p(t)$ is a subgraph of $G_q(t')$.

Property 5 of the above lemma allows us to define $G_p^\infty = \bigcup_{t \in \mathcal{T}} G_p(t)$. From Property 6, we get:

Lemma 19: For any correct processes p and q, $G_p^\infty = G_q^\infty$.

Lemma 19 allows us to define the *limit graph* G to be G_p^∞ for any correct process p. The first four properties of Lemma 18 imply:

Lemma 20: The limit graph G satisfies the four properties of the DAG defined in Section 6.1.

6.5.2 The computation component

Since the limit graph G has the four properties of the DAG, we can apply the "centralized" selection method of Figure 4 to identify a correct process. This method involved:

- Constructing and tagging the infinite simulation forest Υ induced by G.
- Applying a rule to Υ to select a particular correct process p^*.

In the computation component of $T_{D \rightarrow \Omega}$, each p approximates the above method by repeatedly:

- Constructing and tagging the *finite* simulation forest Υ_p induced by G_p, its present finite approximation of G.
- Applying the same rule to Υ_p to select a particular process.

Since the limit of Υ_p over time is Υ, and the information necessary to select p^* is in a finite subgraph of Υ, we can show that *eventually* p will keep selecting the correct process p^*, forever.

Actually, p cannot quite use the tagging method of Figure 4: that method requires knowing which processes are correct! Instead, p assigns tag k to a vertex S in Υ_p^i if and only if S has a descendent S' such that p itself has decided k in $S'(I^i)$. If p is correct, this is eventually equivalent to the tagging method of Figure 4. If p is faulty, we do not care. Also, p cannot use exactly the same selection method as that of Figure 4: its current simulation forest Υ_p may not *yet* have a critical index or contain any deciding gadget (although it eventually will!). In that case, p temporizes by just selecting itself. The computation component of $T_{D \rightarrow \Omega}$ is shown in Figure 6. Let $\Upsilon_p(t)$ denote Υ_p at time t.

Lemma 21: For any correct p and any $t \in \mathcal{T}$:

1. $\Upsilon_p(t)$ is a subgraph[6] of Υ
2. $\Upsilon_p(t)$ is a subgraph of $\Upsilon_p(t+1)$
3. $\lim_{t \rightarrow \infty} \Upsilon_p(t) = \Upsilon$

[6] The subgraph relation ignores the tags.

{*Build and tag simulation forest* Υ_p *induced by* G_p}
for $i \leftarrow 0, 1, \ldots, n$:
  $\Upsilon_p^i \leftarrow$ simulation tree induced by G_p and I^i
  for every vertex S of Υ_p^i
    if S has a descendent S' such that
      p has decided k in $S'(I^i)$
    then add tag k to S

{*Select a process from tagged simulation forest* Υ_p}
if there is no critical index **then return** p
else
  $i \leftarrow$ smallest critical index
  if i is monovalent critical **then return** p_i
  else if Υ_p^i has no gadgets **then return** p
    else return deciding process
      of the smallest gadget in Υ_p^i

Figure 6: Process p's computation component

Lemma 22: For any correct p and any vertex S of Υ_p:

1. p never removes a tag from S.
2. There is a time after which the tags of S in Υ_p will always be the same as the tags of S in Υ.

Theorem 23: For any correct process p, there is a time after which $output_p = p^*$, forever.

Theorem 2: For all environments $\mathcal{E}$, if a failure detector D can be used to solve Consensus in $\mathcal{E}$, then $D \succeq_{\mathcal{E}} \Omega$.

References

[ABND+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, October 1987.

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CT91] Tushar Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[RB91] Aleta Ricciardi and Ken Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351. ACM Press, August 1991.

Improving Fast Mutual Exclusion

Eugene Styer

Department of Math, Statistics and Computer Science
Eastern Kentucky University
Richmond, KY 40475

Abstract

Most mutual exclusion algorithms require $O(n)$ operations to enter the critical section despite how many may be actively trying to enter the critical section. This paper presents a mutual exclusion algorithm that is much more sensitive to how many processes currently want to enter the critical section. This algorithm is based on a parameter l, and assumes there is a total of $n = k^l$ processes. If only one process wants to enter the critical section, $l + 7$ operations are sufficient. If there are t processes currently wanting to enter the critical section, then $O(tlk)$ operations are all that is necessary. This is ususally much less than the $O(n)$ operations required by ordinary mutual exclusion algorithms.

If the need is only to elect one process, the same minimum of 8 operations will hold, but the number of variables used will be $O(\log t)$, if t processes are contending to be elected. This algorithm will also by symmetric, with no distinctions between processes.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0159...$1.50

1 Introduction

The question of mutual exclusion is a very old problem, dating from Dijkstrata in 1965 [2]. However, most of the published solutions require an incoming process to look at every other potential competitor as part of the mutual exclusion process. If the number of such competitors is fairly small this is not a major problem, but some large systems (such as an airline database) could have thousands of processes that might want to examine or change the database. In this case, the time to check for competitors becomes a significant part of the time required for the mutual exclusion problem.

Leslie Lamport [3] came up with a solution to this problem, called Fast Mutual Exclusion. His algorithm allows a process to enter the critical section in a constant number of operations, regardless of the number of potential competitors, since this process is the only one currently attempting to enter the critical section.

Lamport's solution has a problem when there are two (or more) processes seeking the critical section. When two processes start to compete, then it becomes necessary for every process to be checked to identify the competitor. This means the time to enter is either $O(1)$ if you are alone, or $O(N)$ if two processes are trying to enter. The attempt of this paper is to examine solutions to the mutual exclusion problem, and to determine the extent to which the gap between $O(1)$ and $O(N)$ can be narrowed. Ide-

ally we would want the time to be $O(t)$, where t is the number of processes competing for the resource. This paper does not acheive this bound, b· t it does narrow the gap between one process and several processes.

2 Model

The model used in this paper is a variation of the one given in Burns [1]. Let a *system* be a tuple $S = (P, V, Q, \phi, q_0)$, where P is a finite set of processes, V is a finite set of variables, Q is a set of system states, ϕ is a transition function, and q_0 is the specified initial state, with all processes in their remainder section and all variables holding appropriate values. Let $N = |P|$ be the number of processes and $M = |V|$ be the number of global variables in S. We will assign each process has a unique identifier, and assign variables to processes or groups of processes.

2.1 Process State

The state of each process will be represented by a *step* that indicates where that process is within the algorithm. Let X_i be the set of steps in the algorithm that process P_i will take. X_i can be partitioned into R_i, T_i, C_i, and E_i, standing for the remainder region, trying (entry) region, critical section and exit region respectively of process P_i. Also let $X = \bigcup_i X_i$ be the steps of all processes, and define R, T, C and E in a similar manner.

Y_k will be used to represent the set of possible values for variable V_k. By making the assumption that a process can only read or write one variable per step, we can partition the set of steps $T \cup E$ into disjoint sets $Read_k$ and $Write_k$. These sets represent reads and writes respectively. Any other step would represent an internal operation, and is not considered separately. There is one pair of sets $Read_k$ and $Write_k$ for each variable V_k, representing the steps that can read and write that variable. All variables can be read or written by any process, although the algorithm may limit who will use a given variable.

Let $Read = \bigcup_k Read_k$ and $Write = \bigcup_k Write_k$. An *system state* (often just called state) of S is a $(M + N)$-tuple $q = (x_0, x_1, \ldots x_{N-1}, v_1, v_2, \ldots v_M)$, with $x_i \in X_i$ and $v_k \in Y_k$. For each process P_i, x_i is the step it is currently about to execute and v_k is the value of variable V_k. The notation $x_i(q) = x_i$ will indicate the step of a process P_i in state q, and $v_k(q) = v_k$ is the value of the variable v_k in state q. Let Q be the set of all such system states of S.

2.2 Moves and Schedules

Define a move function $\phi: Q \times |P| \to Q$. This is a total function, with $\phi(q, i)$ being the state resulting from initially being in state q, and then letting P_i take one step. A *schedule* is a sequence $h = i_1 i_2 \ldots$ (finite or infinite) of process indices. Define $\phi(q, h)$ in the usual manner by $\phi(q, h) = \phi(\phi(q, i_1), i_2 i_3 \ldots)$. An infinite schedule h is *admissible from* q if no process can stop outside its remainder region. A state q' is *reachable* if $q' = \phi(q, h)$ for some admissible schedule h. For every finite prefix h_1 of h with $x_i(\phi(q, h_1)) \notin R$ there exists a finite prefix $h_1 h_2$ of h such that i occurs in h_2.

The initial state q_0 must conform to the requirements that $x_i(q_0) \in R_i$ for all i.

2.3 Required Conditions

The following conditions enforce our intuitive ideas about deterministic asynchronous systems. For all $q \in Q$, all $i, j \in [0 \ldots N-1]$ and all $q' \in Q$ with $x_i(q) = x_i(q')$ then the following must hold:

1. For all $j \neq i$: $x_j(q) = x_j(\phi(q, i))$.
2. If $x_i(q) \in R \cup T$, then $x_i(\phi(q, i)) \in T \cup C$
3. If $x_i(q) \in C \cup E$, then $x_i(\phi(q, i)) \in E \cup R$
4. If $x_i(q) \in Read_k$ and $v_k(q) = v_k(q')$, then $x_i(\phi(q, i)) = x_i(\phi(q', i))$.

5. If $x_i(q) \notin Read$ then $x_i(\phi(q,i)) = x_i(\phi(q',i))$.

6. if $x_i(q) \in Write_k$, then $v_k(\phi(q,i)) = v_k(\phi(q',i))$.

7. If $x_i(q) \notin Write_k$ then $v_k(q) = v_k(\phi(q,i))$

These conditions enforce the various intuitive expectations that we have for a deterministic asynchronous system. Condition 1 prevents one process' move from affecting any other process. The next two conditions (2 and 3) only permit looping while entering or exiting. Details of the remainder section and critical section are not important here and are suppressed.

Conditions 4 and 5 mean a process can only change its state based on the value read from a variable, and cannot make any choices otherwise. A variable may only change value when a process writes to it (condition 6), and the new value depends only on the state of the writing process. No variable may change value except when some process performs a write to that variable (condition 7).

2.4 Mutual Exclusion

A system S satisfies *mutual exclusion* if for all reachable states $q \in Q$ $X_i(q) \in C$ and $X_j(q) \in C$ imply $i = j$. A system S is *deadlock free* if for all reachable states $q \in Q$ and every admissible schedule h, then for some prefix h' of h either $X_i(\phi(q,h')) \in C$ for some process P_i or $X_i(\phi(q,h')) \in R$ for all i. In particular, schedules involving only processes already in the protocol can continue to make progress. A system S is *lockout-free* if for all reachable states $q \in Q$ and all admissible schedules h from q, then for all processes P_i $X_i(q) \notin R$ implies there is a finite prefix h' of h such that $X_i(\phi(q,h')) \in R$. This prevents a process from being tied up indefinitely while trying to enter or exit the critical section. The conditions 2 and 3 above imply that a process in its trying region must go through the critical region.

3 Algorithm

The algorithm I present is a variation of algorithms by Gary Peterson [4] and Lamport [3], with additional variables to assist in finding any process without having to check every process individually. These algorithms work by having processes first try to enter the critical section by trying a "fast procotol" that uses the minimum number of operations but assumes there is only one process active. If other processes are also be trying to enter the critical section, then a "slow protocol" is entered that uses a more conventional mutual exclusion algorithm to control access to the critical section.

In this algorithm, each process has an assigned variable W_i that has three values: FAST, SLOW and OUT. A process sets W_i to FAST to indicate it is in the fast version of the mutual exclusion protocol. More accurately, FAST means it has not announced it is finished or switching to the slow route. SLOW means the process is trying to enter the critical section, and has already decided that it cannot use the fast protocol due to contention from other processes. OUT means the process has exited the protocol, and is not currently attempting to enter the critical section. Any process P_i that is in its remainder section will have $W_i = OUT$.

Between the individual variables W_k and the global variables that all processes write, there are $l-1$ levels of variables intended to help processes in determing who their competitors are. Assume we have $n = k^l$ total processes (if $n < k^l$, extra 'dummy' processes can be included that only stay in their reminder section). These variables will be called $S_{j,sub(i,j)}$, where j is the current level (1 to $l-1$), and $sub(i,j) = \lfloor i/k^j \rfloor$ (assuming processes are numbered 0 to $n-1$). Each variable in level 1 can be written by any of a group of k processes, each variable in level 2 has an associated group of k^2 processes, and each variable in level m has k^m processes that can write it. At level m, a process i is in group $\lfloor i/k^m \rfloor$.

These variables are written as a process is

entering the protocol, and each process points each variable to itself. These may be overwritten by later processes, but not in a way that can hide a fast route process (proof later). The initial values are not important, and can point to any process in the group.

This method of grouping $(k, k^2, k^3, \ldots)$ in successive levels is also used in the slow mutual exclusion algorithm. To decide the winner of the slow mutual exclusion path, processes first do regular mutual exclusion within their group of k. In level 2 sets of k winners (the ones using each $S_{2,sub(i,2)}$) compete among themselves to choose one of k^2 processes to proceed to level 3. This process (choosing one if k processes) continues for each of the l levels, until one overall winner remains.

Lemma 3.1 *The first violation of mutual exclusion cannot be due to two processes that both took the fast route into the critical section.*

Proof: Suppose two processes P_i and P_j were both simultaneously in the critical section, and both used the fast route to get there. Also suppose P_i was the first of the two to set $Turn := i$. For P_i to take the fast route into the critical section, then P_i must have made the check 'if $turn = i$' before P_j changed $Turn$. However, P_i sets $Lock := true$ as part of entering the critical section. For P_j to continue in the fast route, it must find $Lock$ to be false. Remember A process only clears $Lock$ when exiting the critical section. If P_i cleared $Lock$, then it has finished, and P_j can safely enter the critical section. Any third process P_k clearing $Lock$ would make a potential violation of mutual exclusion (P_i and P_k) prior to the first one to occur, a contradiction. Therefore P_j will find $Lock$ is true, and will turn to the slow route. □

Lemma 3.2 *A process P_i cannot be the first to enter the critical section by the slow route if there is a process P_j capable of entering by the fast route.*

Proof:

```
        W_i := Fast;
        Turn := i;
        if Lock then goto Aside;
        for j := 1 to l − 1 do
            S_{j,sub(i,j)} := i;
        Lock := true;
        if Turn≠ i or Block then goto Aside;
C. S.
        Lock := false;
        W_i := Out;

Aside:  W_i := Slow;
        for j := 0 to l − 1 do
            Entry( j, sub(i,j+1), sub(i,j) );
        Block := true;
        Check_vars( l-1, 0 );
C. S.
        Lock := false;
        Block := false;
        W_i := Out;
        for j := l − 1 downto 0 do
            Exit( j, sub(i,j+1), sub(i,j) );

        Procedure Check_vars( Lev, Start )
        begin
        if Lev > 0 then
          for j := 0 to k − 1 do
              wait until W_{S_{lev,Start+j}} ≠ Fast
              if W_{S_{lev,Start+j}} = Slow then
                Check_vars( Lev-1, (Start+j)*k )
        else
          for j := 0 to k − 1 do
              wait until W_{Start+j} ≠ Fast
        end

        Procedure Entry( level, Vars, MY_id )
        Do normal mutual exclusion entry as
        P_{MY_id} using variables V_{level,Vars}

        Procedure Exit( level, Vars, MY_id )
        Do normal mutual exclusion exit as
        P_{MY_id} using variables V_{level,Vars}
```

Figure 1: Improved Fast Mutual Exclusion Algorithm

- Case 1: P_j has already written its ID to $S_{l-1,sub(j,l-1)}$. Consider the current value of $S_{l-1,sub(j,l-1)}$. If $S_{l-1,sub(j,l-1)} = j$, then P_i will see W_j=FAST, and waits inside of Check_vars until W_j changes to SLOW or OUT. if $S_{l-1,sub(j,l-1)} = t$ with $t \neq j$, then if W_t=FAST, P_i still waits, and if W_t=SLOW, P_i will examine the previous (level $l-2$) set of variables $S_{l-2,x}$ for all processes that might have been hidden by P_t. By a similar argument at level $l-2$, either P_j is visible at this level, or further checking is required. Eventually individual variables will be reached, so P_i will wait for P_j, or for another process.

  The variable W_t cannot only have the value OUT if P_t has already exited the critical section. Process P_t would have had to have written $S_{l-1,sub(j,l-1)}$ after P_j, and by Lemma 3.1 it cannot have taken the fast route, and if P_t took the slow route then P_i would not have the first to be in this situation.

- Case 2: P_j has not written $S_{l-1,sub(j,l-1)}$. In this case, P_i may not discover P_j, but since P_i sets *Block* to true before doing the check, P_j cannot enter the critical section by the fast route because of *Block*. After P_i exits, P_j will again be capable of entering by the fast path, but only after P_i has exited the critical section.

In either case, there cannot be a fast path process in the critical section along with a slow path process. □

Lemma 3.3 *If the mutual exclusion routine that is part of Enter/Exit does not have deadlock or lockout, then this algorithm does not either.*

Proof: Since a process that is still trying the fast route never waits for anything, clearly such a process cannot be waiting indefinitely. Therefore any processes that is waiting indefinitely must be doing so in the slow route. A slow route process first sets W_i to slow and then proceeds through a finite set of mutual exclusion algorithms. By assumption, a process will eventually exit each individual level of the mutual exclusion algorithm. These levels do not interact since each has a separate set of variables, and each level is in the 'critical section' of the next outer level. Once a slow route winner has been chosen, it sets *Block* to true. Therefore any P_j thas has not yet started attempting to enter the critical section will find *Block* is true. So P_j will choose the slow route and be stopped by the mutual exclusion algorithm. Therefore only a finite number of processes can enter the critical section after *Block* was set to true, and the slow route process that set *Block* can enter the critical section. □

Please note that an arbitrary number of fast route processes can pass through the critical section between two successive slow route processes. This can be prevented by having each process checks for other slow route processes before clearing *Block* (unless the Entry/Exit routine allows it). However, an algorithm that has a First-in-first-out (or similar) property loses that property even among the slow route processes.

Lemma 3.4 *Not counting operations done as part of a "***wait until***" loop, at most $O(min(n, tlk))$ operations are required for a slow route process.*

Proof: A count of the variables shows there are $O(n)$ variables, each of which is read or written a constant number of times (excepting the **wait until** loops). We can see this by noting there are n individual variables, $k * O(n/k) = O(n)$ variables for the first level of mutual exclusion routines, $k * O(n/k^2)$ variables in the second round, down to $O(k)$ in the final round, plus a constant number of other variables. Each variable is used a constant number of times (excluding waiting loops). Adding these together shows the $O(n)$ part of the bound.

To show the $O(tlk)$ bound, the main program uses a constant number of operations, the En-

try procedure can be executed l times for $O(k)$ operations each time (assuming a $O(n)$ operation bound for n-process mutual exclusion). The Check_vars procedure does $O(k)$ operations each time it is called. There is one initial call, then if W_m=SLOW Check_vars can be called once for each level l. At most t processes are contenting for the critical section so any other process will show W_m=OUT. Therefore a process can call Check_vars at most $O(tl)$ times, for a total operation bound of $O(tlk)$. □

Theorem 3.1 *The algorithm in Figure 1 maintains mutual exclusion with $l+7$ fast route operations and $O(tlk)$ slow route operations (excluding waits)*

Proof: Mutual exclusion is assured by the combination of Lemmas 3.1, 3.2 and the correctness of the mutual exclusion algorithm embedded in the procedure Entry. The fast route operation count is a simple count of operations (8 fixed, $l-1$ additional), and the slow route count is from Lemma 3.4. □

It should be noted that the correctness of this algorithm does not depend on all of the groups being of the same size. Therefore the size and members of each group can be varied to improve average time if not all processes are expected to attempt to enter the critical section equally often.

Conjecture 3.1 *$O(tlk)$ is a lower bound for this problem.*

My reasoning for this lower bound is that for programs of this type, failing to check all of the earlier $S_{i,j}$ variables would allow a process P_i to be 'hidden' by carefully placed other processes that overwrite key variables, so an incoming process fails to notice P_i. This failure allows P_i to enter by the fast route while another process has entered by the slow route. Allowing the variables to be used in a more general fashion takes a messy situation and confuses it to the point a proof has proven elusive.

4 Fast Election

In considering the problem of election, shorter and simpler algorithms can be found because processes cannot exit and attempt to enter the critical section again. The algorithm and proof are based on the algorithm for symmetric election in Styer and Peterson [5]. This algorithm will use 5 variables and 8 operations in the absence of contention. If t processes are contending, $2\lceil \log t \rceil + 3$ variables are used. If almost every process (t approxmiately equal to n) contends for election, a second upper bound of $2\lceil \log n \rceil - 3$ variables also applies. The number of operations (again excluding waiting loops) will also be $O(\log t)$.

This algorithm is *symmetric*. A symmetric algorithm is one where the only difference between processes is the presence of an identifier that can be compared for equality. The 'variable' *me* will hold that identifier. Every process has the same exact program, and cannot examine two identifiers except to check if they are equal. To phrase this another way, in a symmetric system we can exchange any two processes in a schedule without any third process knowing the difference. Variables can hold an identifier or any of a fixed set of constants. If a variable can hold any identifiers, it must hold them all. Each variable must start with a value that is a constant (typically 0).

To make this more formal, define $x_r(q) \stackrel{ij}{=} x_s(q')$ to mean that two process states $x_r(q)$ and $x_s(q')$ are identical except that wherever P_r has the identifier of P_i, then P_s has P_j's identifier, and vice versa. Then for any schedule h, create the schedule h' by swapping all occurrences of i and j. Symmetry then requires that the result of schedules h and h' must be the same except for i and j.

- For any r not equal to i or j, $x_r(\phi(q_0,h)) \stackrel{ij}{=} x_r(\phi(q_0,h'))$.
- $x_i(\phi(q_0,h)) \stackrel{ij}{=} x_j(\phi(q_0,h'))$.
- $x_j(\phi(q_0,h)) \stackrel{ij}{=} x_i(\phi(q_0,h'))$.

- If $v_m(\phi(q_0, h)) = ID_i$ then $v_m(\phi(q_0, h')) = ID_j$.
- If $v_m(\phi(q_0, h)) = ID_j$ then $v_m(\phi(q_0, h')) = ID_i$.
- If $v_m(\phi(q_0, h))$ is not ID_i or ID_j, then then $v_m(\phi(q_0, h')) = v_m(\phi(q_0, h))$

See Burns [1] and Styer and Peterson [5]) for a more formal definition.

In the algorithm given in figure 2, each variable starts out with the value 0. Then the process that last wrote *turn* attempts to enter the critical section. Other process give priority to the newcomer, and erase any writes they have made. The newcomer waits for each of the variables V_i to be 0, and then writes its own ID into the variable. If it ever notices that *turn* no longer equals itself, it then resets any V_i still holding its ID back to 0. The processes use a parallel set of variables C_i when contention is known to have occurred at this level (two processes simultaneously at or beyond this level). If some V_i has been overwritten by some other process, it is left alone for that other process to handle. The first process can easily enter the critical section. A second process can enter the critical section only the help of a process already at that level (which turns out not to help at all), or itself and one other process at the previous level, which would require $2^{\log n} = n$ processes, but only $n-1$ are available. Also we have a shortcut to election for when only a few processes are active. It will be proved that if level l is reached with no contention visible at level $l-2$, then that process is sufficiently far enough ahead to safely declare itself elected.

The formal proof requires an accounting system of credits to prove mutual exclusion, where particular process and variable states are assigned a value in terms of these credits. To progress to a given point within the algorithm, a process must collect a specified number of credits. Then it is shown that there are not enough credits available for two processes to simultaneously declare themselves elected.

```
turn := me
for level := 1 to ⌈log n⌉ do
    wait until V_level = 0
    V_level := me
    if turn ≠ me then
      for j := 1 to level do
          if V_j = me then
            V_j := 0
          else
            C_j := 1
      Halt
    if C_{level-2} = 0 then
      Announce Elected
Announce Elected
```

Figure 2: $\lceil \log n \rceil + 1$-Variable Symmetric Election Algorithm

Lemma 4.1 *The symmetric election algorithm in Figure 2 has no deadlock.*

Proof: Suppose the system is deadlocked. Since each process only writes *turn* once while entering, there must be a last process to write *turn*, say process P_i. Since P_i cannot make progress, it must be stuck waiting for $V_l = 0$ for some l. Each process can write V_l at most twice (once to *me* and once to 0'), so some process must have been the last to set V_l. At this point, consider the process with the highest value of *level* that has not noticed $turn \neq me$. A process clears all the V's holding its ID as soon as it notices $turn \neq me$, so such a process (call it P_j) must exist. P_j cannot be blocked since there are no higher processes, so it will be able to see $turn \neq me$ and set any variables it has written to 0. Repeat the argument with the new highest process until P_i can proceed. Therefore, this system does not have deadlock. □

Proving that only one process can be elected is not so straightforward, especially as the normal technique of showing each stage eliminates half the processes remaining does not work. Indeed, it is possible to arrange a schedule so that every process sets every variable, reaching the

final test of *turn* before discovering another process has just changed it. The fact that a process can clear several variables, as happens above, would suggest that an exiting process can provide too much help to other processes, and allow a violation of mutual exclusion. In the next lemma we see that this can only happen in limited situations. In particular, it can only happen in place of continuing to the next level. Therefore one process clearing several variables does not cause the processes to violate mutual exclusion.

Each time $V_{level} = 0$ and before it is assigned a nonzero value will be called a *window*. It is only during a window that a process can get past '**while** $V_{level} \neq 0$' and continue to the next level or abort and let others continue. One or more processes may see this window and proceed before any of them sets $V_{level} := me$, but the following lemma limits what can happen afterward.

Lemma 4.2 *For any window on V_{level}, at most one process can change any variable other than V_{level}.*

Proof: Suppose a set of processes $P_1, P_2, \ldots P_K$ all see $V_{level} = 0$ and get past '**while** $V_{level} \neq 0$' before the window closes. One of these (say P_1) must have been the last to set *turn*. If $level = 1$, every other process will see $turn \neq me$, possibly clear V_1, and exit. Therefore assume $level > 1$. Since all of the other processes saw $turn = me$ at the previous level, each of them was beyond the 'if $turn \neq me$' test when P_1 executed $turn := me$. But P_1 reached '**while** $V_{level} \neq 0$' before any of the others could execute $V_{level} := me$, writing $V_1, V_2, \ldots V_{level-1}$ in the process. When the other processes check '**if** $turn \neq me$', it will be true, so these processes will clear any variable holding their id. But in checking $V_1, V_2, \ldots V_{level-1}$, their id has been overwritten (and is never restored), so these processes can only execute $V_j := 0$ for $j = level$. P_1 may continue to the next level or notice $turn \neq me$ and clear some or all of the V_i's, but is the only process that can change any variable other than V_{level}. □

Corollary 1 *For each $V_{level} := 0$, at most one process can go to the next level or clear multiple V_j's.*

Lemma 4.3 *If there are two or more processes simultaneously at or beyond level l, then before the second process arrives the variable C_{l-2} will have already been set to 1.*

Proof: Consider what events can take place at level $l-1$ without C_{l-2} being set. By Lemma 4.2, only one process in any window can continue or find $V_{l-2} = me$. So for any window on V_{l-2} either this process is alone, or any other processes have not yet started clearing $V's$. If a process continues, then V_{l-2} prevents other processes from continuing, and if it stops, then it leaves the algorithm. Either way, there cannot be two processes at level l (but there can be at level $l-1$). Therefore if there are two processes at level l, there must have been at least least one process P_j at level $l-1$ which cannot set V_{l-1} back to 0. This allows P_j to set C_{l-2} to 1, and change V_{l-1}. □

This next lemma is critical in proving mutual exclusion when all N processes are active. It is also used to show the number of variables used (but not correctness) when some process declares itself elected before all the top level is reached.

This lemma introduces the idea of 'credits', which are used here to represent how much progress a given process has made. In order to make progress, other processes had to quit and set various variables V_j back to 0. Each credit will stand for one incoming process, or its equivalent in terms of an initial 0 in some V_j.

Lemma 4.4 *A process must have 2^l credits to reach* **wait until** $V_l = 0$, *and credits are never created.*

Proof: To show that mutual exclusion holds, an accounting system of credits will be set up where a process must collect $2^{level-1}$ credits to reach '**while** $V_{level} \neq 0$'. Every set of actions by one or more processes will maintain the number of credits available, or reduce the number of available credits. Reaching the critical section is equivalent to reaching level $\log n + 1$, and requires n credits. Each process starts out with 1 credit, since every process can reach '**while** $V_1 \neq 0$', for n credits held by processes. We assign each variable V_{level} with $V_{level} = 0$ $2^{level-1}$ credits. The initial 0 values for these variables equals an additional $n-1$ credits, for a total of $2n-1$ credits at the beginning of the algorithm.

Now consider all possible operations by processes:

- A process may fail to execute $V_j := 0$ when it is otherwise capable of doing so. This can happen when a process is slow to check V_j, and its ID is overwritten in the meantime. If this happens, the credits that would have been transferred to V_j are lost. Similarly, if a process writes 0 into a variable already holding 0, those credits are lost.

- Let one or more processes notice $V_{level} = 0$, and let one of them continue to the next level. By Lemma 4.2, the processes that don't go to the next level can only execute $V_{level} := 0$, transferring to V_{level} the $2^{level-1}$ credits they have by getting this far. Then take the $2^{level-1}$ credits from the initial 0 value of the variable, and assign them to the process that continues. This gives it the 2^{level} credits it needs to continue onward.

- Let one or more processes notice $V_{level} = 0$, and let one of them clear some or all of the V_j's. Again by Lemma 4.2, at most one process P_i can clear any V_j other than V_{level}. As above, the remaining processes can transfer their credits by setting $V_{level} := 0$. Again we will assign the credits held by V_{level} to P_i. $P_i$has 2^{level} credits available, half from V_{level} and half from reaching '**while** $V_{level} \neq 0$'. Clearing every variable V_1 to V_{level} accounts for $2^{level} - 1$ credits, losing one (or more) credits. So no credits can be created by a process that clears multiple variables.

The above cases account for all possible actions by processes, so we see that although credits can be lost, they cannot be created. □

Lemma 4.5 *If a process declares itself elected (quick-method), it will be unique.*

Proof: Suppose P_i declares itself elected at level l. By Lemma 4.3 there cannot be another process at this level since P_i found C_{l-2} clear. No processes can get past level l in the future since P_i has set V_l to a nonzero value, and V_l is never cleared back to zero. This prevents any other process from proceeding past **wait until** $V_l = 0$. Also no other process P_j can use the quick exit for election, since this same argument works when we exchange P_i and P_j. If P_j could declare itself elected, then P_i cannot have reached its current position. Therefore if any process P_i finds C_{l-2} clear, it can declare itself elected safely.

□

Theorem 4.1 *The symmetric election algorithm in Figure 2 maintains mutual exclusion, and uses* $\min(2\lceil t \rceil + 3, 2\lceil \log n \rceil - 3)$ *variables if* $t <= n$ *processes participate, and* $O(\log t)$ *operations (exclusing wait loops).*

Proof: Lemma 4.5 proves mutual exclusion when a process uses the quick exit. Otherwise, Lemma 4.4 shows that treating the critical section as level $\lceil \log n \rceil + 1$, at least n credits are necessary for a process to reach the critical section. If two processes were to reach the critical section, $2n$ credits would be necessary. Counting the n credits held by processes and $n-1$ credits from the initial 0 values for the V_j variables, there are $2n-1$ initial credits. However

n credits are required before a process can enter the critical section. So we see that there are not enough credits available for two processes to be in the critical section at the same time, proving mutual exclusion.

As presented, there are $2\lceil \log n \rceil + 1$ variables. Not all of these variables are necessary for the correct and efficient behavior of the algorithm. First, the values of $C_{\log n-2}$ through $C_{\log n}$ are irrelevant. The quick exit is not necessary here since the process will take the normal exit at the end of this level. Also, $C_{\log n-3}$ is not necessary. A process reading $C_{\log n-3}$ is at level $\log n - 1$, so it can simply take the normal exit, modifying one more variable. The algorithm treats these variables as always holding 1.

Next we show that $2\lceil \log t \rceil + 3$ variables are used when only t processes are active. Let $m = \lceil \log t \rceil$. There are $2^m - 1$ credits in V_1 through V_m, and $t \leq m$ credits for initial processes, so there are $2 \cdot 2^m - 1$ credits of interest. Credits stored in higher variables are not counted, since they are either held by some process or returned to that variable. To get two processes to level m requires $2 \cdot 2^m$ credits by Lemma 4.4. Therefore any process checking V_m will always see its own ID, and C_m will remain 0. The discovery that $C_m = 0$ will allow a process at level $m+2$ to take a quick exit. So only *turn*, V_1 through V_{m+2} and C_1 through C_m will ever be referenced, proving the variable bound.

The $O(\log t)$ operation bound is a simple consequence of the fact that processes don't get past level $\lceil \log t \rceil$, and the variables at each level (excluding wait loops) are accesses a maximum of seven times. This counts the initial examination of V_{level}, but not subsequent checks. The check for clearing V_j is included in the cost of level j, although the actual access may take place later. □

We can modify this algorithm to use fewer of the C_j variables. For example, suppose we expect that usually only one process will be active. Then we can only use C_1, with $C_2, C_3, \ldots$ being treated as if they always held the value 1. In general, $\lceil \log t \rceil + 1$ C variables need to be defined to allow a quick exit for t processes. If more than t processes show up, the $C's$ will usually be set to one and the algorithm degenerates to a regular election algorithm.

5 Conclusion

In this paper, we discover that the fast mutual exclusion gap between $O(1)$ operations if a process is alone and $O(n)$ if it is not does not need to be that large. The bound of $O(tlk)$ operations (t is the number of contending processes, l is the number of levels, and $k^l = n$ is the number of processes) is reachable for mutual exclusion.

For one-time election, the bound of $O(\log t)$ provides a very smooth function that is directly related to the number of contending processes, with no jumps or discontinities. This is the same as the mutual exclusion algorithm if alone, but better for small numbers of contending processes.

References

[1] James E. Burns. Symmetry in systems of asynchronous processes. In *22nd FOCS*, pages 169–174. IEEE, 1981.

[2] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, Sept 1965.

[3] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb 87.

[4] Gary L. Peterson, Jan 1988. Private Communication.

[5] Eugene Styer and Gary Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. Technical Report GIT-ICS-89/09, Georgia Institute of Technology, February 1989.

Fast Network Decomposition

(Extended Abstract)

Baruch Awerbuch * Bonnie Berger † Lenore Cowen ‡ David Peleg §

Abstract

This paper obtains the first deterministic sublinear-time algorithm for *network decomposition*. The second contribution of this paper is an in-depth discussion and survey of all existing definitions of network decomposition. We also present a new reduction that efficiently transforms a weak-diameter version of the problem to a strong one. Thus our algorithm speeds up *all* alternate notions of network decomposition. Most importantly for the network applications, we obtain the first fast algorithm for constructing a sparse neighborhood cover of a network, thereby improving the distributed preprocessing time for all-pairs shortest paths, load balancing, broadcast, and bandwidth management.

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM.

†Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by an NSF Postdoctoral Research Fellowship.

‡Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported in part by DARPA contracts N00014-87-K-0825 and N00014-89-J-1988, Air Force Contract OSR-89-02171, Army Contract DAAL-03-86-K-0171 and Navy-ONR Contract N00014-19-J-1698

§Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award.

PoDC '92-8/92/B.C.

© 1992 ACM 0-89791-496-1/92/0008/0169...$1.50

1 Introduction

Sparse neighborhood covers. This paper is concerned with fast deterministic algorithms for constructing sparse neighborhood covers in the distributed network model. Given an undirected (weighted) graph, a *neighborhood cover* is a collection of sets of nodes (also called *clusters*) which cover the neighborhoods of all nodes in the network. A *high-quality* or *sparse* cover (see Section 2) is one that has an optimal tradeoff between the diameter of each cluster and the cluster overlap at single nodes.

The method of representing networks by *sparse neighborhood covers* has recently been identified [13, 12, 3, 15] as the key to the modular design of efficient network algorithms. Using this method as a basic building block leads to dramatic performance improvements for several fundamental network control problems (such as shortest paths [2], job scheduling and load balancing [10], broadcast and multicast [4], deadlock prevention [9], bandwidth management in high-speed networks [7], and database management [15]), as well as for classical problems in sequential computing (such as finding small edge cuts in planar graphs [19] and approximate all-pairs shortest paths [5]). In most of these applications, sparse neighborhood covers yield the first polylogarithmic-overhead solution to the problem. Thus, in a sense, the impact of efficient sparse neighborhood cover algorithms on distributed computing is analogous to the impact of efficient data structures (like balanced search trees or 2-3 trees) on sequential computation.

Our results. This paper presents the first deterministic sublinear-time distributed algorithm (for static synchronous networks) that constructs a high-quality network decomposition. The algorithm runs in time $O(n^\epsilon)$, for any $\epsilon > 0$, improving on the best-known deterministic running time of $O(n \log n)$ for this problem in [13]. We additionally show how to efficiently transform the weak-diameter version of the problem to a strong-diameter sparse 1-neighborhood cover (see Section 2). We get the analogous improvement for t-neighborhood covers, where all algorithmic running times blow up by a factor of t. The distributed algorithm can be adapted to a more realistic dynamic asynchronous environment using the existing transformer techniques in [1, 13, 11]. The applications that use sparse covers as a data structure often run in time $O(t)$, where $t \ll Diam(G) \leq n$, in which case our improvement is particularly significant.

Our results versus existing work. In addition to obtaining the first deterministic sublinear-time algorithm for high-quality network decomposition, we emphasize that we additionally produce clusters that are more useful for applications. The definition of sparse neighborhood covers considered here is equivalent to that in [13], which employs a *strong* notion of the diameter of a cluster (see Section 2). The definition in [13] is related to yet distinct from the notion of network decomposition defined in [8, 18, 16]. Network decomposition as utilized in [8, 18, 16, 17, 6] employs only a *weak* notion of low diameter. This means that the network decomposition clusters might not even be connected within the clusters. Thus they are not sufficient to support, for example, local routing, where the path between two nodes in the same cluster should consist entirely of nodes within that cluster.

We emphasize that the new fast algorithms in this paper speed up *all* alternative notions of network decomposition, including the sparse neighborhood cover definition needed for all the distributed applications. (See Section 2 for precise definitions.)

Other work on related problems. The (weak diameter, low quality) notion of "network decomposition" was first defined in [8, 18]. Awerbuch et. al. [8] gave a fast algorithm for obtaining $O(n^\epsilon)$-diameter clusters, for any $\epsilon > 0$. Their algorithm requires $O(n^\epsilon)$ time in the distributed case, and $O(nE)$ sequential operations. Unfortunately, the construction of [8] is very inefficient in terms of the quality of the decomposition. Roughly speaking, the inefficiency factor is $O(n^\epsilon)$, and this factor carries over to all but some of the graph-theoretic applications, rendering the decomposition of [8] absolutely unacceptable in any practical context. The construction of [8] is, however, sufficient for the two main applications they site in that paper: the maximal independent set problem and $(\Delta + 1)$ coloring. This is because to construct a MIS or a $(\Delta + 1)$ coloring, one needs to traverse the $O(n^\epsilon)$-diameter clusters only a constant number of times. The network control applications, such as routing, online tracking of mobile users, and all-pairs shortest paths, however, need to traverse the clusters many times. A higher-quality decomposition is needed to avoid a large blowup in the running time for these latter applications.

For the remainder of this paper, when we refer to network decomposition, we mean any of the formulations of high-quality decomposition, and not the large diameter, large number of colors obtained by [8].

Subsequent to our work, Pasconesi and Srinivasan [17] slightly reduced the running time for the poor-quality and weak-diameter construction in [8]. While [8] obtained a running time of $O(n^\epsilon)$, where $\epsilon = O(\sqrt{\log \log n}/\sqrt{\log n})$, [17] reduced ϵ to $\tilde{\epsilon} = O(1/\sqrt{\log n})$. As a consequence of the better running time achieved in [17] (smaller ϵ) and the techniques in this paper, the running time of our algorithm for high-quality network decomposition can be slightly improved (see Corollary 3.4).

The randomized algorithm of Linial and Saks [16] achieves a high-quality decomposi-

tion by introducing randomization. The results are stated in terms of a weak notion of low-diameter clusters, but can be modified to produce a strong diameter decomposition as well, using, for example, the reduction techniques in this paper (or [11]). The resulting algorithm is efficient. (We comment that the distributed algorithm is valid only for static synchronous networks, but the efficient transformer techniques of [11] extends it to a (more realistic) dynamic asynchronous model.) Since we are concerned here with using neighborhood covers as a data structure and running various applications on top, a randomized solution is not acceptable in many cases. We need a fast deterministic algorithm that *guarantees* a good underlying neighborhood cover.

2 Definitions

Notions of network decomposition. We survey the different formulations of network decomposition, and discuss their relations. Within each family of definitions, we also discuss what it means to have a *high-quality* decomposition or cover, in terms of the optimal tradeoffs between low diameter and sparsity. The sparse neighborhood cover formulation is the one that is useful for all the applications. We stress that the algorithms in this paper achieve all alternative notions of network decomposition.

Definition 2.1 Consider a graph G whose vertices appear in sets $S_1, \ldots, S_r$. The *weak* distance between $u, v \in S_i$, denoted $dist_G(u,v)$, is the length of the shortest path between u and v in G. Namely, the path is allowed to shortcut through vertices not in S_i. The *weak diameter* of S_i, $diam(S_i) = \max_{u,v \in S_i}(dist_G(u,v))$

Definition 2.2 Consider a graph G whose vertices appear in sets $S_1, \ldots, S_r$. The *strong* distance between $u, v \in S_i$, denoted $dist_{S_i}(u,v)$, is the length of the shortest path between u and v, on the induced subgraph S_i of G. Namely, all vertices on the path connecting u and v are also in S_i. The *strong diameter* of S_i, $Diam(S_i) = \max_{u,v \in S_i}(dist_{S_i}(u,v))$.

The *square* of a graph, G^2, is defined to be the graph G with additional edges if there exists a w s.t. (u,w) and (w,v) are in G. Similarly, G^t is the graph with an edge between any two vertices that are connected by a path of length $\leq t$ in G. The *j-neighborhood* of a vertex $v \in V$ is defined as $N_j(v) = \{w \mid dist_G(w,v) \leq j\}$. Similarly, the j-neighborhood of a set, V, is defined to be $N_j(V) = \cup_{v \in V} N_j(v)$,

We are now ready to define the alternate notions of network decomposition. First, we give the weak diameter definition.

Definition 2.3 For an undirected graph $G = (V,E)$, a *(χ, d, λ)-decomposition* is defined to be a χ-coloring of the nodes of the graph that satisfies the following properties:

1. each color class is partitioned into an arbitrary number of disjoint *clusters;*
2. the *weak* diameter of any *cluster* of a single color class is at most d.
3. clusters of the same color are at least distance $\lambda+1$ apart.

A *(χ, d, λ)-decomposition* is said to be *high-quality* if when $d = O(k\lambda)$, χ is at most $kn^{1/k}$.

We make several remarks about the Definition 2.3, which is equivalent to the definitions in [8, 16, 17].

- The high-quality decomposition as defined above achieves the optimal tradeoff; there are graphs for which χ must be $\Omega(kn^{1/k})$ to achieve a decomposition into clusters of diameter bounded by $O(k\lambda)$ and separation λ [16].
- When $\lambda = 1$, we will abbreviate this as a (χ, d)-decomposition. Typically, we are most concerned with the case of a high-quality decomposition when χ and d are both $O(\log n)$. This optimal decomposition tradeoff is not achieved in the clusters of [8, 17], but is achieved by randomized methods in [16]. The algorithms in

this paper are the first to achieve the optimal tradeoff deterministically in sublinear time.

- The main application known for this structure in "symmetry-breaking"– it can be used to construct a maximal independent set or a $\Delta+1$ coloring fast in the distributed domain [8, 16, 17]. In this paper, we use the structure for symmetry breaking as follows: the recursive algorithm in Section 3 constructs a (χ, d, λ)-decomposition on the power of the graph inside the recursion first, and later uses this to obtain a *strong-diameter* decomposition.

For *strong-diameter* network decomposition, the definition is the same as Definition 2.3, except in Step 2, substitute strong for weak diameter. As with the weak definition, the "high-quality" tradeoffs are optimal. A *strong-diameter* (χ, d)-decomposition can be thought of as a generalization of the standard graph coloring problem, where χ is the number of colors used, and the clusters are supernodes of diameter d.

We now present the definition for sparse neighborhood covers. Notice that this is a strong diameter definition.

Definition 2.4 A (k,t)-*neighborhood cover* is a collection of sets (also called *clusters*) of nodes $S_1, \ldots, S_r$, with the following properties:

1. $\forall v, \exists i$ s.t. $N_t(v) \subseteq S_i$, where $N_t(v) = \{u | dist_G(u,v) \leq t\}$.
2. $\forall i, Diam(S_i) \leq O(kt)$, where $Diam(S_i) = \max_{u,v \in S_i}(dist_{S_i}(u,v))$.

A (k,t)-neighborhood cover is said to be *sparse*, if each node is in at most $kn^{1/k}$ sets.

Setting $k = 1$, the set of all balls of radius t around each node is a sparse neighborhood cover. Setting $k = Diam(G)/t$, the graph G itself is a sparse neighborhood cover. In the first case, the diameter of a ball is t, but each node could appear in every ball. In the second case, each node appears only in G, but the diameter of G could be as high as n. Setting $k = \log n$ (the typical and useful setting, for all the applications), a sparse $(\log n, t)$-neighborhood cover is a collection of sets S_i with the following properties: the sets contain all t-neighborhoods, the diameter of the sets is bounded by $O(t \log n)$, and each node is contained in at most $c \log n$ sets, where $c > 0$. We remark that this bound is tight to within a constant factor; there exist graphs for which any $(\log n, t)$-neighborhood cover places some node in at least $\Omega(\log n)$ sets [16]. When $k = \log n$, we find that sparse neighborhood covers form a useful data structure to *locally* represent the t-neighborhoods of a graph.

Our new fast distributed algorithm achieves deterministically a structure which is simultaneously a (strong, and therefore also weak) diameter decomposition *and* a sparse neighborhood cover.

3 Weak Diameter Network Decomposition

In this section, we introduce the new distributed algorithm `Color`, which recursively builds up a $(kn^{1/k}, 2k, 1)$-decomposition. It calls on a procedure, `Create_New_Color`, which runs a modified version of the Awerbuch-Peleg [14] greedy algorithm on separate clusters.

Note that all distances in the discussion below, including those in the same cluster, are assumed to be *weak* distances, and the diameter of the clusters is always in terms of weak diameter (see Section 2).

`Color` is implicitly taking higher and higher powers of the graph, where recall that we define the graph G^t to be the graph in which an edge is added between any pair of nodes that have a path of length $\leq t$ in G. Notice that to implement the graph G^t in a distributed network G, since the only edges in the network are still the edges in the underlying graph G, to look at all our neighbors in the graph G^t, we might have to traverse paths of length t. Therefore the time

for running an algorithm on the graph G^t, blows up by a factor of t. The crucial observation is that a $(\chi, d, 1)$-decomposition on G^t is a (χ, dt, t)-decomposition on G. Choosing t well at the top level of the recursion, guarantees that nodes in different clusters of the same color are always separated by at least twice the maximum possible distance of their radii. We can thus use procedure `Greedy_Create_Color` to in parallel *recolor* these separate clusters without collisions. (The leader of each cluster does all the computation for its cluster.)

The recursive algorithm has two parts:

1. Find a (χ, dt, t)-decomposition, where $\chi = xkn^{1/k}$, $d, t = 2k$, on each of x disjoint subgraphs.
2. Merge these together by recoloring, as just described, to get a $(kn^{1/k}, 2k, 1)$-decomposition.

Algorithm: `Color`(G)

Input: graph $G = (V, E)$, $|V| = n$, and integer $k \geq 1$.
Output: A $(kn^{1/k}, 2k, 1)$-decomposition of G.

1. Compute G^{2k}.
2. If G has less than x nodes, run the Linial-Saks [16] or Awerbuch-Peleg [14] simple greedy algorithm on G^{2k}, and go to step 6.
3. Partition nodes of G into x subsets, $V_1, \ldots, V_x$ (based on the last $\log x$ bits of node IDs, which are then discarded).
4. Define G_i to be the subgraph of G^{2k} induced on V_i.
5. In parallel, for i, `Color`(G_i).
   (every node of G is now colored recursively)
6. For each $v \in V$, color v with the color $<i, \text{color}(v) \in G_i>$.
   (this gives an $xkn^{1/k}$ coloring of G with separation $2k$)
7. Do sequentially, for $i = 1$ to $kn^{1/k}$, `Create_New_Color`(G, i)
   (this gives a $kn^{1/k}$ coloring of G with separation 1)

Algorithm: `Create_New_Color`(G, i)
(this colors a constant fraction of the old-colored nodes remaining with new color i)

Input: graph G with new and old colored nodes such that there is a $(xkn^{1/k}, (2k)^2, 2k)$-decomposition on the old-colored nodes of G and a $(i-1, 2k, 1)$-decomposition on the new-colored nodes of G
Output: graph G with new and old colored nodes such that there is a $(xkn^{1/k}, (2k)^2, 2k)$-decomposition on the old-colored nodes of G and a $(i, 2k, 1)$-decomposition on the new-colored nodes of G

1. $W \leftarrow V$.
2. Do sequentially, for $j = 1$ to $xkn^{1/k}$, "Look at nodes with old color j":
   (a) Do in parallel for color j clusters,
   - Elect a leader for each cluster.
   - The leader learns the identities, U, of all the nodes in W within k distance from the border of its cluster (i.e. this is graph G for that cluster).
   - The leader calls procedure `Greedy_Create_Color`(R, U), where R is the set of old-colored j nodes in both the leader's cluster and in W.
   - `Greedy_Create_Color` returns (DR, DU). The leader colors the nodes in DR with new color i, and sets $W \leftarrow W - DU$.

`Greedy_Create_Color` is the procedure of the Awerbuch-Peleg [14] greedy algorithm that determines what nodes will be given the current new color. The algorithm identifies a constant fraction of the nodes in the cluster R to be colored. The algorithm picks an arbitrary node in R (call it a *center* node) and greedily grows a ball around it of minimum

radius r, such that a constant fraction of the nodes in the ball lie in the interior (i.e. are in the ball of radius $r-1$ around the center node). It is easy to prove that there always exists an $r \leq k|R|^{1/k}$ for which this condition holds. Note that although the centers of the balls grown out are always picked (arbitrarily) from the nodes in R, the interiors and borders of the balls which are then claimed, include any of the nodes in U (not just those in R) within the ball. Then another arbitrary node is picked, and the same thing is done, until all nodes in R have been processed. Procedure `Create_New_Color` will then color the interiors of the balls (set DR) with new color i, and remove each entire ball from the working graph W.

Algorithm: `Greedy_Create_Color`(R, U)

Input: sets of nodes R and U, where R is the set of nodes in the cluster and U is a superset of nodes that contains R.

Output: (DR, DU). This returns a constant fraction of the nodes in R in set DR and the 1-neighborhoods of the clusters of DR in set DU.

1. $DR \leftarrow \emptyset$; $DU \leftarrow \emptyset$.
2. While $R \neq \emptyset$ do
   (a) $S \leftarrow \{v\}$ for some $v \in R$.
   (b) While $|N_1(S) \cap U| > |R|^{1/k}|S|$ do $S \leftarrow S \cup (N_1(S) \cap U)$.
   (c) $DR \leftarrow DR \cup S$.
   (d) $DU \leftarrow DU \cup (N_1(S) \cap U)$.
   (e) $R \leftarrow R - S - (N_1(S) \cap R)$.
   (f) $U \leftarrow U - S$.

Lemma 3.1 If $x = 2^{\sqrt{\log n}\sqrt{1+\log k}}$, the running time of the procedure `Color` is $n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2$.

Proof The branching phase of the recursion takes time $T'(n) \leq 2kT'(n/x)+x$. The merge takes time $\chi k n^{1/k} td = x(kn^{1/k})^2(2k)^2$, where $\chi k n^{1/k}$ is the number of iterations overall and td is the number of steps per iteration. Overall, we have

$$\begin{aligned} T(n) &\leq 2kT(n/x) + x(kn^{1/k})^2(2k)^2 \\ &\leq (2k)^{\log n/\log x} x(kn^{1/k})^2(2k)^2 \\ &\leq n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2, \end{aligned}$$

when $x = 2^{\sqrt{\log n}\sqrt{1+\log k}}$. □

Theorem 3.2 There is a deterministic distributed asynchronous algorithm which given a graph $G = (V, E)$, finds a $(kn^{1/k}, 2k, 1)$-decomposition of G in $n^{2\sqrt{1+\log k}/\sqrt{\log n}+2/k}(2k)^2$ time.

Corollary 3.3 There is a deterministic distributed asynchronous algorithm which given $G = (V, E)$, finds a $(O(\log n), O(\log n), 1)$-decomposition of G in $n^{O\left(\sqrt{\log\log n}/\sqrt{\log n}\right)}$ time, which is n^{ϵ} for any $\epsilon > 0$. We remark that the constant on the big-oh in the running time is 3.

As a corollary to our theorem and [17], we can obtain a slightly better running time.

Corollary 3.4 There is a deterministic distributed asynchronous algorithm which given $G = (V, E)$, finds a $(O(\log n), O(\log n), 1)$-decomposition of G in $O(n^{1/\sqrt{\log n}})$ time, which is n^{ϵ} for any $\epsilon > 0$.

4 Strong Diameter Network Decomposition

The algorithm in the previous section produced a weak-diameter network decomposition. While this is a nice problem, the strong-diameter form is the one we want in order to successfully run most distributed applications. In this section, we give a reduction that given a weak diameter decomposition, constructs a structure that is simultaneously both a strong diameter decomposition and a sparse neighborhood cover. The algorithm as written outputs the cover: the associated strong decomposition consists of the interiors of the clusters in the sparse neighborhood cover.

We introduce an algorithm **Sparse**, which takes as input a procedure **Decomp**, which given a graph $G = (V, E)$, finds a $(kn^{1/k}, 2k, 1)$-decomposition of G. In actuality, we will bind **Decomp** to procedure **Color** of Section 3. **Sparse** first calls procedure **Decomp** with G^{8kt}. Of course, this will yield an $O(kt)$ blowup in the running time of **Decomp**, say τ.

Once **Decomp** is called, the remaining running time for **Sparse** is $O(k^2 n^{2/k})$, times a t blowup for traversing t-neighborhoods. Then, in sum, **Sparse** is able to obtain a t-neighborhood cover in the original graph G in time $O(kt\tau + k^2 t n^{2/k})$. Recall that k is typically $\log n$.

Notice that the code for **Sparse** is similar to the last pass of procedure **Color** (Section 3); however, **Sparse** has an additional level of complexity. To obtain a t-neighborhood cover, we must modify the Awerbuch-Peleg [13] coarsening algorithm, called as a subroutine, so that we can recolor clusters in parallel without interference.

Notation. In the algorithms below, we use roman capital letters for names of sets, and calligraphic letters for names of collections of sets. In particular, corresponding to a set W, by convention we will denote by $\mathcal{W}$ the collection consisting of the sets $\{N_t(v) | v \in W\}$.

Algorithm: **Sparse**(G,**Decomp**)

Input: graph $G = (V, E)$, $|V| = n$, and integer $k \geq 1$, and a procedure **Decomp**, that finds a $(kn^{1/k}, 2k, 1)$-decomposition of G.
Output: $\mathcal{T}$, a sparse (k, t)-neighborhood cover of G.

1. **Decomp**(G^{8kt}).
   (*returns a $(kn^{1/k}, 2k, 1)$-decomposition of G^{8kt} which is a $(kn^{1/k}, 16k^2 t, 8kt)$-decomposition of G.*)
   (a) $\mathcal{T} \leftarrow \emptyset$.
   (*$\mathcal{T}$ is the cover.*)
   (b) Do sequentially, for $i = 1$ to $kn^{1/k}$,
   (*find a $kn^{1/k}$-degree t-neighborhood cover of G.*)
   i. $\mathcal{U} \leftarrow \{N_t(v) | v \in V\}$.
   (*$\mathcal{U}$ is the collection of all unprocessed t-neighborhoods.*)
   ii. Do sequentially, for $j = 1$ to $kn^{1/k}$,
   "Look at nodes with old color j":
   A. Do in parallel for color j clusters,
   - Elect a leader for each cluster.
   - The leader learns the identities, of all the t-neighborhoods of nodes within a $4kt$ distance from the border of its cluster.
   - The leader calls procedure **Cover**($\mathcal{R}$,$\mathcal{U}$) on G, where $\mathcal{R}$ is the collection of t-neighborhoods of old-colored j nodes in both the leader's cluster and in U.
   - **Cover** returns $(\mathcal{DR}, \mathcal{DU})$. The leader colors the nodes in $\mathcal{DR}$ with new color i, and sets $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{DU}$.
   iii. $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{DR}$

Cover is our modification of the Awerbuch-Peleg [13] coarsening algorithm that determines what nodes will be given the current new color. The actual code for this procedure follows a description of the algorithm below. The key to our fast simulation of their coarsening algorithm, is that we keep track of neighborhoods within and outside of the old-colored j clusters separately, in order to recolor clusters in parallel without collisions.

Procedure **Cover**($\mathcal{R}$,$\mathcal{U}$) operates in iterations. Each iteration constructs one output cluster $Y \in \mathcal{DT}$, by merging together some clusters of $\mathcal{U}$. The iteration begins by arbitrarily picking a cluster S in $\mathcal{U} \cap \mathcal{R}$ and designating it as the kernel of a cluster to be constructed next. The cluster is then repeatedly merged with intersecting clusters from

$\mathcal{U}$. This is done in a layered fashion, adding one layer at a time. At each stage, the original cluster is viewed as the internal kernel Y of the resulting cluster Z. The merging process is carried repeatedly until reaching a certain sparsity condition (specifically, until the next iteration increases the number of clusters merged into Z by a factor of less than $|\mathcal{R}|^{1/k}$). The procedure then adds the kernel Y of the resulting cluster Z to a collection $\mathcal{DT}$. It is important to note that the newly formed cluster consists of only the kernel Y, and not the entire cluster Z, which contains an additional "external layer" of $\mathcal{R}$ clusters. The role of this external layer is to act as a "protective barrier" shielding the generated cluster Y, and providing the desired disjointness between the different clusters Y added to $\mathcal{DT}$.

Throughout the process, the procedure keeps also the "unmerged" collections $\mathcal{Y}, \mathcal{Z}$ containing the original $\mathcal{R}$ clusters merged into Y and Z. At the end of the iterative process, when Y is completed, every cluster in the collection $\mathcal{Y}$ is added to $\mathcal{DR}$, and every cluster in the collection $\mathcal{Z}$ is removed from $\mathcal{U}$. Then a new iteration is started. These iterations proceed until $\mathcal{U} \cap \mathcal{R}$ is exhausted. The procedure then outputs the sets $\mathcal{DR}$ and $\mathcal{DT}$.

Procedure **Cover** is formally described in Figure 1. Its properties are summarized by the following lemma. We comment that our modifications do not change the lemma.

Lemma 4.1 ([13]) Given a graph $G = (V, E)$, $|V| = n$, a collection of clusters $\mathcal{R}$ and an integer k, the collections $\mathcal{DT}$ and $\mathcal{DR}$ constructed by Procedure Cover$(\mathcal{R}, \mathcal{U})$ operates in iterations. satisfy the following properties:

(1) All clusters in $\mathcal{DR}$ have their t-neighborhood contained in some cluster in $\mathcal{DT}$.

(2) $Y \cap Y' = \emptyset$ for every $Y, Y' \in \mathcal{DT}$,

(3) $|\mathcal{DR}| \geq |\mathcal{R}|^{1-1/k}$, and

(4) $\max_{T \in \mathcal{DT}} Diam(T) \leq (2k-1) \max_{R \in \mathcal{R}} Diam(R)$.

$\mathcal{DT} \leftarrow \emptyset$; $\mathcal{DR} \leftarrow \emptyset$
repeat
  Select an arbitrary cluster $S \in \mathcal{U} \cap \mathcal{R}$.
  $\mathcal{Z} \leftarrow \{S\}$
  repeat
    $\mathcal{Y} \leftarrow \mathcal{Z}$
    $Y \leftarrow \bigcup_{S \in \mathcal{Y}} S$
    $\mathcal{Z} \leftarrow \{S \mid S \in \mathcal{U},\ S \cap Y \neq \emptyset\}$.
  until $|\mathcal{Z}| \leq |\mathcal{R}|^{1/k} |\mathcal{Y}|$
  $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{Z}$
  $\mathcal{DT} \leftarrow \mathcal{DT} \cup \{Y\}$
  $\mathcal{DR} \leftarrow \mathcal{DR} \cup \mathcal{Y}$
until $\mathcal{U} \cap \mathcal{R} = \emptyset$
Output $(\mathcal{DR}, \mathcal{DT})$.

Figure 1: Procedure Cover$(\mathcal{R}, \mathcal{U})$.

Theorem 4.2 There is a deterministic distributed algorithm, e.g. Sparse$(G$,Color$)$, that given a graph $G = (V, E)$, $|V| = n$, and integers $k, t \geq 1$, constructs a t-neighborhood cover of G in $k^2 t n^{O(\sqrt{\log k}/\sqrt{\log n})}$ time in the asynchronous model, where each node is in at most $O(k n^{1/k})$ clusters, and the maximum cluster diameter is $O(kt)$.

Finally, we remark that if we color only the *interiors* of the new color i clusters, the above construction produces a strong diameter high-quality network decomposition from a weak diameter high-quality network decomposition, as well as a sparse neighborhood cover. This is because our construction is such that each node in the cover lies in precisely one new-colored interior.

5 Acknowledgment

Thanks to Tom Leighton for helpful discussions.

References

[1] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, Oct. 1987.

[2] Y. Afek and M. Ricklin. Sparser: A paradigm for running distributed algorithms. Unpublished manuscript, 1991.

[3] Y. Afek and M. Riklin. Sparser: A paradigm for running distributed algorithms. *J. of Algorithms*, 1991. Accepted for publication.

[4] B. Awerbuch, A. Baratz, and D. Peleg. Efficient broadcast and light-weight spanners. Unpublished manuscript, Nov. 1991.

[5] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Fast deterministic cover algorthms. Unpublished manuscript, Nov. 1991.

[6] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Low-diamter graph decomposition is in NC. In *Proc. 3'rd Scandinavian Workshop on Algorithm Theory*, July 1992. to appear.

[7] B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan, and S. Kutten. Distributed control for paris. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 145–160, 1990.

[8] B. Awerbuch, A. Goldberg, M. Luby, and S. Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, May 1989.

[9] B. Awerbuch, S. Kutten, and D. Peleg. On buffer-economical store-and-forward deadlock prevention. In *Proc. of the 1991 INFOCOM*, 1991.

[10] B. Awerbuch, S. Kutten, and D. Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 571–580, 1992.

[11] B. Awerbuch, B. Patt, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks with polylogarithmic overhead. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 557–570, 1992.

[12] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[13] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[14] B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM J. Disc. Math*, 5(2):151–162, 1992.

[15] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 39–50, 1992.

[16] N. Linial and M. Saks. Decomposing graphs into regions of smali diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, Jan. 1991.

[17] A. Pasconesi and A. Srinivasan. Improved algorithms for network decompositions. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 581–592, 1992.

[18] D. Peleg. Distance-preserving distributed directories and efficient routing schemes. unpublished manuscript, 1989.

[19] S. Rao. Finding small edge cuts in planar graphs. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 229–240, 1992.

Leader Election in Complete Networks

Gurdip Singh
Department of Computing and Information Sciences,
Kansas State University, Manhattan, KS 66506
singh@cis.ksu.edu

Abstract: This paper presents protocols for leader election in complete networks. The protocols are message optimal and their time complexities are a significant improvement over currently known protocols for this problem. For asynchronous complete networks with sense of direction, we propose a protocol which requires $O(N)$ messages and $O(logN)$ time. For asynchronous complete network without sense of direction, we show that $\Omega(N/logN)$ is a lower bound on the time complexity of any message optimal election protocol and we present a family of protocols which requires $O(Nk)$ messages and $O(N/k)$ time, $logN \leq k \leq N$. Our results also improve the time complexity of several other related problems such as spanning tree construction, computing a global function, etc.

1 Introduction

In the leader election problem, there are N processors in the network, each having a unique identity. Initially all nodes are *passive*. An arbitrary subset of nodes, called the *base* nodes, wake up spontaneously and start the protocol.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0179...$1.50

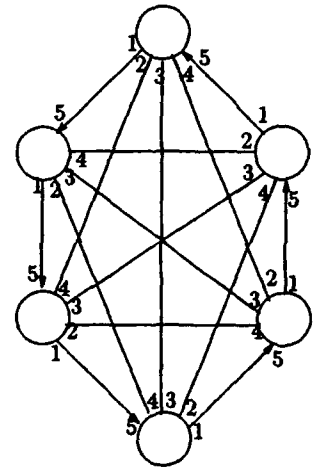
On the termination of the protocol, exactly one node announces itself the leader. In this paper, we consider the problem of electing a leader in an asynchronous complete network. In a complete network, each pair of nodes is connected by a bidirectional link and we assume that a node is initially unaware of the identity of any other node.

Leader election is a fundamental problem in distributed computing and has been studied in various computation models. For complete networks in which a node is unable to distinguish between its incident links, [KMZ84] showed that $\Omega(NlogN)$ messages are required for electing a leader. However, [LMW86] showed that the lower bound of $\Omega(NlogN)$ messages does not hold for complete networks with sense of direction and gave a protocol which requires $O(N)$ messages. A network has a *sense of direction* if there exists a directed Hamiltonion cycle and each edge incident at any node i is labeled with the distance of the node at the other end along this Hamiltonion cycle. Figure 1 shows a complete network containing six nodes with a sense of direction. [ALSZ89] further showed that $O(logN)$ chords in a ring network are sufficient to obtain a protocol with $O(N)$ message complexity. These two extreme cases, one in which a node is unable to distinguish between any two incident edges and the other in which all edges are labeled with a distinct number, show the impact of knowledge of topological information on

the complexity of leader election.

The time complexity of the protocol in [LMW86] is $O(N)$. In this protocol, a node captures a majority of nodes before it declares itself as the leader. We observe that in such networks, a node does not have to capture a majority of nodes in order to be elected the leader. We use this idea to obtain a simple protocol which requires $O(N)$ messages. However, due to congestion on the links and a specific wake up pattern of nodes, its time complexity is not $O(logN)$. We then modify this protocol to solve these problems and obtain a protocol which requires $O(N)$ messages and $O(logN)$ time.

We also propose an improved protocol for leader election in asynchronous complete network without sense of direction (in the rest of the paper, unless other stated, we will use 'complete network' to mean 'complete network without sense of direction'). [KMZ84] proposed a protocol for this problem which requires $O(NlogN)$ messages and $O(NlogN)$ time. [AG85] gave a series of simple message optimal protocols for complete networks, each with $O(N)$ time complexity. Furthermore, it was conjectured in [AG85] that $\Omega(N)$ is a lower bound on the time complexity of any message optimal election protocol for asynchronous complete networks. We prove that $\Omega(N/logN)$ is a lower bound on the time complexity of any message optimal protocol for this problem. This proves that introducing asynchrony may result in a loss in speed by a factor of $N/(logN)^2$. A similar result was shown in [AFL83] where a particular asynchronous system was shown to be slower by a factor of $logN$ than the corresponding synchronous system. We also provide a message optimal protocol for asynchronous complete networks which requires $O(N/logN)$ time. The protocol involves a new technique which allows us to distinguish between nodes that wake up at different times to participate in the protocol. The complexity of the protocol depends on the number of base nodes and we show that the time complexity can be improved to $O(logN + min(r, N/logN))$, where r is the number of base nodes. We also present a protocol tolerant to f initial site failures which requires $O(Nf + NlogN)$ messages and $O(N/logN)$ time, where $f < N/2$.



The directed Hamiltonian cycle

Figure 1: A complete network with a sense of direction

There are many problems such as spanning tree construction, computing a global function, etc. which are equivalent to leader election in terms of message and time complexities. Our protocols, therefore, leads to improvement in the time complexity of these problems as well.

This paper is organized as follows. In the next section, we present our model of distributed computation. In Section 3, we present a protocol for leader election in a complete network with sense of direction. In Section 4, we present a protocol for leader election in a complete network without sense of direction and we show a lower bound on the time complexity of any message optimal leader election protocol.

2 Model

We model the communication network as a complete graph (N, E), where N and E represent the processors and communication links respectively. We assume that each node has a unique identity. Messages sent over a link arrive at their destination within finite but unpredictable time and in the order sent and are not lost. Each message may carry $O(logN)$ bits of information. The message complexity of a protocol is the maximum number of messages sent during any possible execution of the protocol. The time complexity of a protocol is the worst case execution time assuming that each message takes at most one time unit to reach its destination and computation time is negligible. Furthermore, inter-message delay on a link is at most one time unit. All additions in the paper are assumed modulo N.

3 Complete Networks with sense of direction

For complete networks with sense of direction, [LMW86] proposed a leader election protocol which requires $O(N)$ messages and $O(N)$ time. Let $i[d]$ denote the node at distance d from i and $i[x..y]$ denote the set $\{i[x], i[x+1], \ldots, i[y]\}$. In [LMW86], if node i is able to capture nodes in $i[1..N/2]$ then it can declare itself as the leader. We observe that in the presence of a sense of direction, a node does not have to capture a majority of nodes. For example, if i captures all nodes in $\{i[1..N/4], i[N/2], i[3N/4]\}$ then it can declare itself as the leader. By capturing $i[N/2]$, for example, i ensures that no node in $i[N/4+1..N/2]$ will be able to become a leader since a node in this set must capture $i[N/2]$ to become a leader. In particular, a node can declare itself as the leader after capturing the nodes in the set $\{i[1..k], i[2k], i[3k], \ldots, i[N-k]\}$. We combine this idea with those in [LMW86] to obtain a new protocol, $\mathcal{A}$, which is as follows:

Protocol $\mathcal{A}$: This protocol proceeds in two phases:

First Phase: On waking up spontaneously, a base node i tries to capture nodes in $S_i = i[1..k]$ in a sequential fashion. A *passive* node wakes up on receiving a message of the protocol. A *passive* node is not allowed to become a base node if it wakes up on receiving the message of the protocol. A base node i uses its identity and $level_i$, which denotes the number of nodes which i has captured so far, to contest with other nodes. When a base node i wakes up, it sends a message $capture(i, level_i)$ to $i[1]$. When a node j receives a $capture(i, l)$ messages, it behaves as follows:

- If j is not a base node or it has been captured then it responds with an $accept(0)$ message.
- If j is a base node which has not yet been captured, and $(level_j, j) < (l, i)$ then again, i captures j and j responds with $accept(level_j)$. Otherwise, j ignores the message.

If i receives $accept(l)$, it adds $l+1$ to $level_i$ (and therefore the set of captured nodes is extended to include the nodes captured by j). If $level_i < k$ then it continues its conquest by sending a *capture* message to $i[level_i+1]$. Otherwise, it enters the second phase.

Second Phase: On entering this phase, i sets $owner_i$ to i and sends a message, $owner(i)$, to each node j in $i[1..k]$. On receiving this message, j sets $owner\text{-}link_j$ to denote the link from j to i and $owner_j$ to i. Furthermore, it sends an acknowledgement message to i. After receiving an acknowledgement from all nodes in $i[1..k]$, i sends an $elect(i)$ message to each node in $\{i[2k], \ldots, i[N-k]\}$. On receiving $elect(i)$, site

j behaves as follows: If ($owner_j$ has not been set) or ($owner_j$ has been set and $owner_j < i$) then it sets $owner_j$ to i and sends an *accept* message to i. Otherwise, it ignores the message. If i receives all accept responses then it declares itself the leader.

In the first phase, each node is captured at most once and each capturing requires one *capture* message and one *accept* message. Hence, the first phase will require $O(N)$ messages. In the second phase, there can be at most $O(N/k)$ candidates. Each candidate sends messages to capture N/k nodes. Hence, the total number of messages in the second phase is $O(N^2/k^2)$. The message complexity of $\mathcal{A}$ is therefore $O(N + N^2/k^2)$. In particular, for $k \geq \sqrt{N}$, the protocol requires $O(N)$ messages. We will now compute the time complexity of $\mathcal{A}$. The execution of the second phase takes $O(1)$ time. Furthermore, if a node is successful in capturing another node then it does in a constant amount of time. Hence, the node which is elected the leader will finish its first phase within $O(k)$ time units of waking up. However, the following situation can arise: Assume that nodes have identities $1, \ldots, N$ such that $i[1] = i + 1$. Let 1 be the first node to wake up. After waking up spontaneously, node i sends a *capture* message to node $i + 1$. $i + 1$ wakes up just before the message from i reaches it and sends the message to capture $i + 2$ before receiving the message from i. In this case, no response will be sent to i since i has the same level number as $i+1$ but a smaller identity. If this happens for all sites i, $1 \leq i \leq N$, then only node N will survive and capture all other nodes. If the capture message for each node takes exactly one time unit to arrive, node N will wake up at time $N - 1$ and therefore the protocol will require $O(N)$ time units. However, *if all nodes wake up within $O(k)$ time of each other then the first phase will take $O(k)$ time.* We will now modify the protocol $\mathcal{A}$ to obtain $\mathcal{A}'$ as follows: After a node i wakes up (either spontaneously or on receiving a message), it sends a message to awaken $i[1]$ and $i[k]$. Hence, within $O(k + N/k)$ time, all nodes will wake up to participate in the protocol. Therefore, the time complexity of the protocol is $O(k + N/k)$. In particular, for $k = \sqrt{N}$, the time complexity of $\mathcal{A}'$ is $O(\sqrt{N})$.

We will now extend this idea to obtain a protocol which requires $O(logN)$ time. Consider the following protocol, $\mathcal{B}$, which is an asynchronous version of the synchronous protocol in [AG85]. For simplicity, assume that N is of the form 2^x. In this protocol, a candidate node i tries to capture all other nodes in $logN$ steps. In the first step, i sends a message to capture $i[N/2]$. In the l^{th} step, i sends a message to capture $2^l - 1$ nodes in the set $i[N/2^l], i[3N/2^l], \ldots, i[(2^l - 1)N/2^l]$. If $i[N/2]$ is also a base node then it will send a message to capture i in its first step. Hence, only one of i and $i[N/2]$ will proceed to step 2. Similarly, only one of i, $i[N/4]$, $i[N/2]$ and $i[3N/4]$ will proceed to step 3 and so on. Although the time complexity of this protocol is $O(logN)$, its message complexity is $O(NlogN)$.

We will now combine ideas in $\mathcal{A}$ and $\mathcal{B}$ to obtain a protocol $\mathcal{C}$ which has $O(N)$ message complexity and $O(logN)$ time complexity. $\mathcal{C}$ proceeds in two phases. In the first phase, we use $\mathcal{A}$ to first reduce the number of candidates to at most $N/logN$. The second phase employs $\mathcal{B}$ to elect the leader. Let $k = N/2^{\lceil loglogN \rceil}$.

First Phase: In this phase, i tries to capture $i[k], i[2k], \ldots, i[N - k]$ in a sequential manner. Observe that when $i[xk]$ wakes up, it will try to capture the same set of nodes in the order $i[xk + k], \ldots, i[xk + N - k]$. Hence, in the first phase, nodes in this set compete against each other. The rules for capturing are the same as in the first phase of $\mathcal{A}$. Hence, for example, if $i[xk]$ has already captured $i[(x + 1)k]$ when i captures $i[xk]$, then $i[xk]$ surrenders $i[(x + 1)k]$

to i and therefore, i can extend its set of captured nodes to include this node also. Using i as the reference node, the nodes can be partitioned into k sets, $R_0, \ldots, R_{k-1}$, where $R_j = \{i[j+k], i[j+2k], \ldots, i[j+N-k]$. After the first phase, we have at most one alive candidate from each of these sets.

Second Phase: On entering this phase, node i sends messages to each node j in R_i to update $owner_j$ to i. In this phase, we have at most one candidate in each set R_j, and we have to elect a leader among them. Let i be the candidate in R_0. In order to defeat other candidates, i tries to capture the nodes in the set $i[1..k-1]$ in $logk$ steps. This phase is an asynchronous version of the synchronous protocol in [AG85]. In the first step, i sends an $elect(i,0)$ message to capture $i[k/2]$. In the l^{th} step, it sends 2^{l-1} messages to capture $i[k/2^l], i[3k/2^l], \ldots, i[(2^l-1)k/2^l]$. Observe that if there is an alive candidate in $R_{k/2}$ then it will send a message to capture a node in R_0 in its first step. Hence, only one node from R_0 and $R_{k/2}$ will go to step 2. In step 2, since a node sends messages to capture nodes at distance $k/4$ and $3k/4$, only one node from R_0, $R_{k/4}$, $R_{k/2}$ and $R_{3k/4}$ will survive. In general, after the l^{th} step, there will be at most $k/2^l$ alive candidates (note that k is $O(N/logN)$). In this phase, to contest with other nodes, i uses its identity and $step_i$, which indicates the number of steps which i has executed so far. Consider the case in which a node, i, in R_x sends a message to capture a node, j, in R_y. If j is passive then there is no base node in R_y which has captured all nodes in R_y and therefore, j sends an *accept* message to i. If there is a candidate in R_y then the message is forwarded to that node ($owner\text{-}link_j$ will be the edge leading to this node) and they compete on the basis of $(step, id)$. However, if i's message reaches the candidate in R_y and finds that this node has already been captured then i must first kill the owner of R_y's candidate before it can claim R_y's candidate. For this purpose, the message is forwarded to that node (thus, each message can be forwarded at most twice). For example, if i in $R_{k/2}$ sends a message to capture j in $R_{3k/4}$, and the base node in $R_{3k/4}$ has already been captured by $R_{k/4}$ in step 1 then i must defeat the base node in $R_{k/4}$ before claiming $R_{3k/4}$.

The first phase requires $O(logN)$ time since a node competes only with $O(logN)$ other nodes. The second phase involves $O(logN)$ steps, each of which will take a constant amount of time. Hence, the protocol requires $O(logN)$ time. We will now compute the message complexity of $\mathcal{C}$. The first phase requires $O(N)$ messages since a node is captured at most once. In the second phase, there can be at most k candidates. Furthermore, *there can be at most* $k/2^{l-1}$ *nodes in step* l (since a node in step l must have captured 2^{l-1} nodes and sets of nodes captured by different sites are disjoint). A node in step l sends 2^{l-1} messages to capture nodes. Each of these messages generate a constant number of messages. Since $k = O(N/logN)$, the total number of messages generated in the second phase is $\sum_{1 \le l \le logN}(k/2^{l-1} * O(2^{l-1}))$ $= \sum_{1 \le l \le logN}(O(N/(logN * 2^{l-1})) * O(2^{l-1})) \le$ $O(logN * N/logN) = O(N)$. Hence, the message complexity of the protocol is $O(N)$.

4 Complete Networks without sense of direction

In this section, we will present a family of algorithms for leader election in complete networks without sense of direction. Protocols belonging to this family require $O(Nk)$ messages and $O(N/k)$ time, where $logN \le k \le N$ [Si91]. We will first present two different algorithms, $\mathcal{D}$ and

$\mathcal{E}$, for leader election. We will then combine features of these algorithms to obtain the final protocol $\mathcal{F}$.

Protocol $\mathcal{D}$: In this algorithm, a *base* node attempts to capture all other nodes in parallel. On waking up spontaneously, a base node sends its identity in an *elect* message on all incident edges. When a node j receives an *elect*(i) message over edge e, it behaves as follows: If j is a base node and $j > i$ then no response is sent over e; Otherwise, j sends an *accept* message over e. A node that receives an *accept* message on all incident edges declares itself the leader. The time complexity of this protocol is O(1). However, its message complexity is $O(N^2)$ since the number of base nodes may be $O(N)$, each of which will send $O(N)$ messages.

Protocol $\mathcal{E}$: $\mathcal{E}$ is a modification of the protocol **A** in [AG85]. The outline of protocol **A** in [AG85] is as follows:

> A base node tries to capture other nodes in a sequential manner by sending *capture* messages on its incident edges one at a time. A node that is successful in capturing all other nodes is elected the leader. A base node i sends its identity and a variable, $level_i$, in the *capture* message to contest with other nodes ($level_i$ is the number of nodes which i has captured so far). If a capture message from i reaches a node j which has not yet been captured, and $(level_i, i) \geq (level_j, j)$ (lexicographically) then i captures j, otherwise i is killed. If j is a captured node, then i has to kill j's owner before claiming j. If i is successful in capturing j then it increments $level_i$ and proceed with its conquest by sending a capture message to another node.

The message complexity of **A** is $O(NlogN)$. Although the time complexity of **A** is $O(N)$, it does not possess the property that a node is able to capture another node in a constant amount of time. For example, a captured node j may receive *capture* messages from nodes $i_1, i_2, \ldots, i_m$ in the order given and forward each of these messages to its owner. In particular, if it forwards $O(N)$ messages and only the last forwarded message is able to defeat $owner_j$ then it may take $O(N)$ time to capture j (since the messages are forwarded on the same link and inter-message delay on the same link can be 1 time unit, the last forwarded message may reach j after $O(N)$ time units). We modify **A** to obtain $\mathcal{E}$ in which there is at most one forwarded message on a link at any time. In $\mathcal{E}$, a captured node j uses a boolean variable $forward_j$ to keep track of whether or not it has forwarded a message to its owner. If j receives a capture message and $forward_j$ is true then it delays forwarding the message to its owner until it receives a response from its owner. Each message forwarded to the owner is responded by an accept or a reject message depending on whether the forwarded message defeated the owner. If an accept message is received then j sends an accept message to the node from which it has received the largest $(level, id)$ pair so far. If a reject message is received, j forwards the message with the largest $(level, id)$ pair it may have received in the meanwhile. Thus, in $\mathcal{E}$, if a node is able to capture another node then it does so in a constant amount of time.

In $\mathcal{D}$, if the number of candidates is restricted to $O(k)$ then it will require $O(Nk)$ messages. In protocol $\mathcal{E}$, there can be at most k nodes at level N/k [AG85]. We obtain a new protocol $\mathcal{F}$ in which $\mathcal{E}$ is used to reduce the number of candidates for protocol $\mathcal{D}$ by requiring a node to execute $\mathcal{E}$ until its level number reaches N/k and $\mathcal{D}$ thereafter, where $logN \leq k \leq N$. The protocol $\mathcal{F}$ is as follows:

> On waking up spontaneously, a node starts executing protocol $\mathcal{E}$. When a node reaches level N/k, it sends an *elect* message with its identity on all incident edges. Let node

j receive an *elect*(i) message over e. If $(level_j, maxid_j)$ is less than $(N/k, i)$ then j changes status to *killed* and sends an *accept* message over e. A node which receives an *accept* message on all incident edges declares itself the leader.

Since the message complexity of $\mathcal{E}$ is $O(NlogN)$ and at most k nodes broadcast an *elect* message, the message complexity of $\mathcal{F}$ is $O(Nk)$ (since $k \geq logN$). Since it takes a constant amount of time to capture a node, it will take $O(N/k)$ time for a node to reach level N/k after it wakes up (if it reaches this level). After a node reaches this level, it executes $\mathcal{D}$ which takes $O(1)$ time. Therefore, the node which is elected the leader will take $O(N/k)$ time after waking up spontaneously to declare itself the leader. Thus, we have the following lemma:

Lemma 4.1 *If all nodes wake up within $O(N/k)$ time of each other, then $\mathcal{F}$ will terminate in $O(N/k)$ time.*

However, a situation similar to the one in the first-phase of protocol $\mathcal{A}$ for networks with sense of direction (in which $i+1$ wakes up just before the message from i reaches it) can occur which may lead to an execution in which the node which is elected the leader wakes up $O(N)$ time units after the first node wakes up. Since nodes are unable to distinguish between the incident edges, the solution used in the presence of sense of direction will not work here. However, we also have the following result: After a node i reaches level k, only a node at level at least k can capture it. Hence, in every interval of c time units after a node reaches level k, where c is a constant, either the node with the highest level number (which is greater than k) will increase its level number or it will be killed by another node with level number at least k. Since there are at most N/k nodes at level k, some node will reach level N/k within $2cN/k$ time. Thus, we have the following lemma:

Lemma 4.2 *After a node reaches level k, $\mathcal{F}$ terminates within $O(N/k)$ time.*

In the following, we will design a protocol, $\mathcal{G}$, in which we will ensure that *in every interval of c time units, either at least k nodes wake up or some node reaches level at least k.* Then from Lemma 4.1 and Lemma 4.2, the protocol will require $O(N/k)$ time. For this purpose, we require a base node to execute two initial phases on waking up spontaneously. If it successfully executes these phases, it qualifies as a candidate for election and proceeds by executing $\mathcal{F}$. Intuitively, the time complexity of the leader election protocol depends on the ability to recognize the order in which nodes wake up to participate in the protocol so that nodes that wake up later are prohibited from becoming candidates. In the first phase, a node tries to obtain permission from k other nodes. If i requests permission from j after j has finished executing its first phase, it denies permission to i. In this case, i gets ordered after j and is not allowed to participate as a candidate. However, as we will show later, this allows a node which wakes up $O(N/k)$ time units after the first node wakes up to obtain permission from k other nodes and participate as a candidate. The first-phase is as follows:
On waking up spontaneously, node i selects k incident edges and sends a *first-phase*(i) message on each of these edges. On receiving this message, site j behaves as follows:

- If j is not a captured node then
  if j has finished executing its first phase then it sends a *finish* message over e.
  ⋆ If j is *passive* then i becomes j's owner and j marks e as *owner-link*$_j$. It sends an *accept* message over e and changes its state to *captured*.
  ⋆ If j is in the first phase then j sends a *proceed* message over e.
- If j is *captured* then it checks whether its owner has finished the first phase. For

this purpose, it sends a *check* message over *owner-link*$_j$ (if it has already sent a check message, it waits for the reply to avoid congestion). If the owner replies that it has finished the first phase, then it sends a *finish* message to i and to any other node from which it has received (or will receive in the future) a *first-phase* message in the meantime. If the owner has not finished the first phase then j sends a *proceed* message to i and also to any other node from which it may have received a message in the meantime.

After node i has received responses to all k *first-phase* messages, it behaves as follows: It exits the first phase. If it has received a *finish* message then it does not enter the second phase and changes status to *killed*. Otherwise, it enters the second phase. It also updates its level number to the number of *accept* messages received in the first phase. In the second phase, node i tries to reach level k. For this purpose, it sends a *capture*$(level_i, i)$ message on each edge on which it received a *proceed* message. The rules for capturing are the same as in protocol $\mathcal{E}$ with the following changes: Nodes which have not started the second phase are regarded as *passive* by these capture messages. A node increases its level number only after receiving an *accept* response to each *capture* message sent in the second phase. After finishing the second phase, a node executes $\mathcal{F}$.

In $\mathcal{G}$, a base node finishes its first phase within 5 time units of waking up [Si92]. Furthermore, there will be a node which enters the second phase and a node which finishes the second phase to participate in $\mathcal{F}$. Hence, the protocol will elect a leader. Using the technique in [BKWZ87], we also extend our protocol to obtain a protocol resilient to f initial site failures, where $f < N/2$, which requires $O(Nf + NlogN)$ messages and $O(N/logN)$ time.

Lemma 4.3 *The time complexity of* $\mathcal{G}$ *is* $O(N/k)$.

Proof: A base node will finish its first phase within 5 time units of waking up. If node i wakes up spontaneously at time t then for i to participate in the second phase, each of its *first-phase* messages must go to a node which is in its first phase (this node must have awakened spontaneously after time $t-5$, otherwise it will have finished its first phase by time t) or is *passive* (this node will wake up by time $t+1$ as a result of i's message). Therefore, i is able to proceed to the second phase only if at least k nodes other than i wake up in the interval $[t-5, t+1]$.

Consider an interval of 5 time units, say $[m, m+5]$ where $m \geq 0$, during the execution of the protocol. We have the following cases:
(1) At least $k+1$ nodes wake up in the interval $[m-5, m+6]$.
(2) Less than $k+1$ nodes wake up in the interval $[m-5, m+6]$. In this case, we will show that some node will reach level at least k. All nodes that wake up before time m will finish their first phase by time $m+5$. Any node which completes its first phase in the interval $[m+5, m+6]$ will not be able to participate in $\mathcal{F}$ as a candidate (since it must have awakened in the interval $[m, m+6]$ and less than $k+1$ nodes wake up in the interval $[m-5, m+6]$). If a node has not already reached level k then let i be the node with the highest identity among the nodes which are in the second phase at time $m+5$. Let a *capture* message from i reach a node j which is not captured. We have the following cases:
(a) If j has not started the second phase then it will respond with an *accept* message.
(b) If j has started the second phase then it must be the case that j entered the second phase at or before time $m+5$ and therefore $j < i$ (by assumption). Hence, j will respond with an *accept* message.

If j is a captured node then it will send forward the message to its owner. If the $owner_j$ has not started second phase then it will send an *accept* message. Otherwise, we will show that $owner_j$ must have entered the second phase before time $m+5$. Assume not. Since nodes that wake up before time m enter the second phase before time $m+5$ and nodes that wake in the interval $[m, m+5]$ do not participate in the second phase, $owner_j$ must have awakened after time $m+5$. In this case, the *first-phase* message from $owner_j$ will reach j after time $m+5$. However, by this time, i would already have sent its *first-phase* message to j and therefore, $owner_j$ cannot capture j. Hence, $owner_j$ must have finished the second phase before time $m+5$ in which case it will send an *accept* message to i since $owner_j < i$ by assumption. Thus, i will receive all *accept* responses and therefore i will finish its second phase.
Hence, in each interval of 11 time units ($m-5$ to $m+6$), either at least k nodes wake up or some node will reach level k. Therefore, by time $11N/k$, either all nodes will be awake or some node will have reached level k. Then, from Lemma 4.1 and Lemma 4.2, the protocol will terminate in $O(N/k)$ time units. □

In [Si92], we show that the time complexity of $\mathcal{G}$ depends of the number of candidate nodes. By using the capturing pattern of the synchronous protocol in [AG85], we have obtained a message optimal protocol which requires $O(logN + min(r, N/logN))$, where r is the number of candidate nodes.

5 A Lower Bound

We will now prove that $\Omega(N/logN)$ is a lower bound on the time complexity of any message optimal protocol for leader election in complete asynchronous networks. We will restrict ourselves to *comparison-based* leader election algorithms. We prove the following theorem:

Theorem 5.1 *Any comparison-based protocol for leader election in a complete asynchronous network which sends less than Nd messages will require at least $N/16d$ time.*

Corollary 5.1 *Any message optimal protocol for leader election in a complete asynchronous network, i.e., requiring $O(NlogN)$ messages, will require $\Omega(N/logN)$ time.*

Proof of Theorem 5.1: For simplicity, assume that nodes have identities belonging to the set $\{1, \ldots, N\}$ and let $k = 2d$. A one-to-one function f from a set of processor identities to another set of processor identities is *order-preserving* if $i \leq j$ implies $f(i) \leq f(j)$. Two lists $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_m\}$ are *order-equivalent* if $(x_i \leq x_j) \Leftrightarrow (y_i \leq y_j)$. Following [FL87], we assume that each processor's local state consists of its identity, its initial state and the history of messages it has received so far. Further, a node sends its entire local state in each message. Intuitively, a process state includes all events that can potentially affect this state [Lam78] and the *happens before* ordering information between these events. Let $event(i,t)$ denote the set of events which can potentially affect the state of i at time t in an execution. We say that $event(i,t)$ and $event(j,t')$ are *order-equivalent* if there exists an order-preserving function which maps $event(i,t)$ to $event(j,t')$ such that the happens-before relation is preserved. If $event(i,t)$ and $event(j,t')$ are order-equivalent then we say that the state of i at time t and state of j at time t' are order-equivalent. A comparison-based protocol cannot distinguish between order-equivalent states. Our proof involves showing that we can keep processes in order-equivalent states for a long period of time.

Let A be a protocol for leader election which requires Nd messages. Let Ex be the set of executions of A in which (1) all nodes wake up spontaneously at the same time, (2) all messages take ϵ time to reach their destination, where $\epsilon < 1/2$ and (3) if a set of messages arrive at the same time at a site then the messages are accepted in the increasing order of sender identities. Let Up_i denote the ordered list of edges from i to nodes $i+1, \ldots, i+k$, arranged in the increasing order of sites identities and $Down_i$ denote the set of edges from i to nodes with identities $i-1, \ldots, i-k$. Since a node cannot distinguish between untraversed incident links, the adversary has the freedom to choose any untraversed edge whenever the node wants to send a message over an untraversed edge. In particular, the adversary acts as follows: Whenever a node i has to send a message over a new edge, the adversary selects the edges first from Up_i. If all edges in Up_i have been used then it selects other unused edges. The actions of the adversary try to impose a symmetry on the nodes. We partition the nodes into the following sets: $\{S_1, \ldots, S_{N/k}\}$, where $S_i = \{k(i-1)+1, \ldots, ki\}$. Let $R = S_2 \cup \ldots \cup S_{m-1}$, where $m = N/k$ and $R' = S_1 \cup S_m$. As long as nodes in R remain in order-equivalent states, each node i will only communicate with nodes in $Up_i \cup Down_i$; otherwise each node in R will send messages over at least $k+1$ edges and the number of messages will exceed Nd. This fact and conditions (1)-(3) impose a symmetry on nodes in R. Intuitively, nodes in R cannot break symmetry without communicating with nodes in R'. Each node i in R executes the same protocol and communicates with at most k nodes with larger identities (belonging to Up_i) and at most k nodes with smaller identities (belonging to $Down_i$). However, nodes in R' are asymmetric with respect to these nodes. In any execution, let $nodes(i,t)$ denote the set of sites at which events in $event(i,t)$ occur. Then, for example, at time ϵ, sites 1 and i, where $i \in R$, may not be in order-equivalent states since 1 may receive a message from a node with a higher identity while i only receives messages from nodes with lower identities. However, at this time, $event(i,\epsilon)$ and $event(j,\epsilon)$, where $i, j \in R$, are order-equivalent. Hence, any node i in R will only send messages to nodes in $Up_i \cup Down_i$ at time ϵ. Observe that $nodes(i,\epsilon) \subseteq S_{y-1} \cup S_y \cup S_{y+1}$, where $i \in S_y$. If 1 sends a message at time ϵ to site $i \in S_2$ then it can force i and another site $j \in R$ to be in order-inequivalent states at time 2ϵ. However, for each node $i \in S_3 \cup \cdots \cup S_{m-2}$, if $nodes(i,2\epsilon) \subseteq S_{y-2} \cup \cdots \cup S_{y+2}$ then nodes in $S_3 \cup \cdots \cup S_{m-2}$ will be in order-equivalent states at time 2ϵ (in this case, no node would have received a message at time 2ϵ from a node in R' which, at time ϵ, was in a state order-inequivalent with respect to states of nodes in R). In general, we prove the following: Let $M_x = S_{x+1} \cup \cdots \cup S_{m-x}$ and $depth(i,t)$ denote the longest chain of messages involving events in $event(i,t)$.

Lemma 5.1 *There exists an execution in Ex such that at any time t and $i \in S_y \cap M_x$, where $x \leq m/4$, if $depth(i,t) \leq x$ and $nodes(i,t) \subseteq S_{y-x} \cup \cdots \cup S_{y+x}$, then for all $j \in M_x$, $event(i,t)$ and $event(j,t)$ are order-equivalent.*

Proof Outline: We prove this by induction on x. The result is immediate for $x = 0$ since processes are initially in order-equivalent states. Assume that the hypothesis holds for $x \leq l$. Then there exists an execution in Ex such that at any time t, if $depth(i,t) \leq l$ and $nodes(i,t) \subseteq S_{y-l} \cup \cdots \cup S_{y+l}$ then for all $j \in M_l$, $event(i,t)$ and $event(j,t)$ are order-equivalent. Let t be the maximum time in this execution at which $depth(i,t) \leq l$ for $i \in M_l$ (*i.e.*, any message sent after this time increases the depth). Assume that at some time t' in this execution, $depth(i,t') = l+1$ and $nodes(i,t') \subseteq S_{y-l-1} \cup \cdots \cup S_{y+l+1}$, where $i \in M_{l+1}$. Let $j \in M_{l+1}$. Since $i, j \in M_l$, $event(i,t)$ and $event(j,t)$ are order-equivalent. Let p send a message to i at time t which is in $event(i,t')$ but not in $event(i,t)$. Then $depth(p,t) = l$ and

$p \in M_l$ (otherwise, we can show a contradiction since $i \notin Up_p \cup Down_p$). Let $q = j-(i-p)$. Then $q \in M_l$. From the induction hypothesis, $event(p,t)$ and $event(q,t)$ are order-equivalent. Hence, q will also send a message to j. Therefore, $event(i,t')$ and $event(j,t')$ will be order-equivalent. □

Lemma 5.2 *There exists an execution in Ex such that at any time t and $i \in M_{m/4} \cap S_y$, if $nodes(i,t) \subseteq S_{y-m/4} \cup \cdots \cup S_{y+m/4}$ and $depth(i,t) \leq m/4$ then i must have communicated only with nodes in Up_i and $Down_i$.*

Proof: Assume not. Assume that i sends a message to a node not in $Up_i \cup Down_i$. Let $j \in M_{m/4}$. Then by Lemma 5.1, there exists an execution in which $event(i,t)$ and $event(j,t)$ are order-equivalent. Since i sends a message to a node not in $Up_i \cup Down_i$, j will also send a message to a node not in $Up_j \cup Down_j$ (since order-equivalent states are indistinguishable to a comparison-based protocol). Thus, each node in $M_{m/4}$ will send at least $k+1$ messages. Since $|M_{m/4}| = N/2$, the execution will involve at least $N(k+1)/2$ messages which is a contradiction. □

We will make use of symmetry between nodes in R to construct an execution in which nodes in R' wake up much later in the protocol to break symmetry. Let ex be an execution of A. Let the nodes be partitioned into three sets, P_1, P_2 and P_3 such that $P_1 = \{1,\ldots,p_1\}$, $P_2 = \{p_1+1,\ldots,p_2\}$ and $P_3 = \{p_2+1,\ldots,N\}$. Assume that (1) nodes in P_1 and P_3 wake up at time t, (2) no messages have been sent by nodes in P_2 to nodes in P_1 and P_3 up to time t and (3) all messages sent at or after time t take ϵ time units. Let $g(ex,P_2)$ denote the execution in which all nodes in P_2 wake up $1-\epsilon$ time earlier than in ex but all links incident on nodes in P_2 remain idle in the interval $[t-(1-\epsilon),t]$ *i.e.*, transmission of messages does not make progress during this period. Due to the asynchronous nature of the network, no node can distinguish between ex and $g(ex,P_2)$. This technique of increasing message transmission time without violating the happens before ordering is similar to the one in [AFL83].

Let M be the set of messages in ex sent at time t from i to j, where $i,j \in P_2$ and j receives no messages from nodes in $P_1 \cup P_2$ at time $t+\epsilon$. Let $h(ex,P_2)$ denote the execution in which links incident on nodes in P_2 are not idle in the period $[t-(1-\epsilon),t]$ but all messages sent in this interval except those in M take $\epsilon+(1-\epsilon)$ time. The effect of this transformation is to increase the delays on the links from ϵ to $\epsilon+1-\epsilon$, which cannot be distinguished from links remaining idle for $1-\epsilon$ time units. Hence, ex and $h(ex,P_2)$ are indistinguishable to all nodes.

We will construct a set of executions in a series of steps which takes $O(N/k)$ time. Let ex be an execution in Ex which satisfies Lemma 5.2. Let $A_1 = S_1 \cup \cdots \cup S_{m/2-1}$, $B_1 = S_{m/2}$ and $C_1 = S_{m/2+1} \cup \cdots \cup S_m$, where $m = N/k$. All nodes wake up at time $t = 0$ and no messages are sent before that time. Furthermore, all messages take ϵ time units. Hence, ex and $h(ex,B_1)$ are indistinguishable to all nodes. Let $ex_1 = h(ex,B_1)$. Let $A_2 = S_1 \cup \cdots \cup S_{m/2-2}$, $B_2 = S_{m/2-1} \cup S_m \cup S_{m/2+1}$ and $C_2 = S_{m/2+2} \cup \cdots \cup S_m$. In ex_1, no messages are sent to nodes in A_2 and C_2 before time $1-\epsilon$ (since nodes in $S_{m/2}$ communicate only with nodes in $S_{m/2-1}$ and $S_{m/2+1}$ until nodes not in R' wake up (From Lemma 5.2)). Further, all messages sent at or after time $1-\epsilon$ take ϵ time units. Therefore, ex_1 and $h(ex_1,B_2)$ are indistinguishable to all nodes. Since ex_1 and ex are indistinguishable, ex and $h(ex_1,B_2)$ are also indistinguishable to all nodes. Let $ex_2 = h(ex_1,B_2)$. Continuing in this way, we can construct an execution $ex_{m/4}$, where $A_{m/4} = S_1 \cup \cdots \cup S_{m/4}$, $B_{m/4} = S_{m/4+1} \cup \ldots \cup S_{3m/4}$ and $C_{m/4+1} = S_{3m/4} \cup \cdots \cup S_m$, which is indistinguishable from ex. The execution time of $ex_{m/4}$ is at

least $m/4(1-\epsilon)$. Since $m/4(1-\epsilon) = N/4k(1-\epsilon) = N/8d(1-\epsilon) \geq N/8d(1/2) = N/16d$, we have an execution which requires $N/16d$ time. □

6 Conclusion

In this paper, we have presented distributed algorithms for leader election in complete networks. For asynchronous networks with a sense of direction, we first presented a simple protocol which requires $O(N)$ messages and $O(\sqrt{N})$ time. We then improved the time complexity of this protocol to $O(logN)$ time. An interesting question is whether synchronized clocks can be used to improve the time complexity of this protocol.

For completer asynchronous networks without sense of direction, we also showed a lower bound of $\Omega(N/logN)$ on the time complexity of any message optimal leader election protocol. We presented a protocol which requires $O(N/logN)$ time and $O(NlogN)$ messages. In [AG85], a lower bound of $\Omega(logN)$ on the time complexity of any message optimal protocol for synchronous complete networks was shown. This proves that introducing asynchrony may result in a loss in speed by a factor of $N/(logN)^2$. In [Si92], we study the problem of leader election in partially synchronous networks and present lower bounds for such networks.

References

[AFL83] Arjomandi, E., Fischer, M., and Lynch, N. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3), 1983.

[AG85] Afek, Y. and Gafni, E. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of ACM Sym. on Principles of Distributed Computing*, 1985.

[ALSZ89] Attiya, H., Leeuwen, J., Santoro, N., and Zaks, S. Efficient elections in chordal ring networks. *Algorithmica*, 4, 1989.

[BKWZ87] Bar-Yehuda, R., Kutten, S., Wolfstahl, Y., and Zaks, S. Making distributed spanning tree algorithms fault-resilient. In *STACS*, 1987.

[FL87] Frederickson, G. and Lynch, N. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proceedings of ACM Sym. on Theory of Computing*, 1987.

[KMZ84] Korach, E., Moran, S., and Zaks, S. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of ACM Sym. on Principles of Distributed Computing*, 1984.

[Lam78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communciation of the ACM*, 21(7), July 1978.

[LMW86] Loui, M., Matsushita, T., and West, D. Election in complete networks with a sense of direction. *Info. Processing Letters*, 22, April 1986.

[Si91] Singh, G. Efficient distributed algorithms for leader election. In *IEEE International Conference on Distributed Computing Systems*, 1991.

[Si92] Singh, G. Upper and lower bounds for leader election in complete networks. In *Technical Report 92-8, Kansas State University*, 1992.

The Impact of Time on the Session Problem

Injong Rhee Jennifer L. Welch
Department of Computer Science
CB 3175 Sitterson Hall
University of North Carolina at Chapel Hill
Chapel Hill, N.C. 27599-3175

Abstract

The *session* problem is an abstraction of synchronization problems in distributed systems. It has been used as a test-case to demonstrate the differences in the time needed to solve problems in various timing models, for both shared memory (SM) systems [2] and message-passing (MP) systems [4]. In this paper, the session problem continues to be used to compare timing models quantitatively. The session problem is studied in two new timing models, the *periodic* and the *sporadic*. Both SM and MP systems are considered. In the periodic model, each process takes steps at a constant unknown rate; different processes can have different rates. In the sporadic model, there exists a lower bound but no upper bound on step time, and message delay is bounded. We show upper and lower bounds on the time complexity of the session problem for these models. In addition, upper and lower bounds on running time are presented for the semi-synchronous SM model, closing an open problem from [4]. Our results suggest a hierarchy of various timing models in terms of time complexity for the session problem.

1 Introduction

Early work in distributed computing usually assumed one of two extreme timing models: either the completely synchronous model, in which processes operate in lockstep rounds of computation, or the completely asynchronous, in which there are no upper bounds on process step time or message delay. Since both of these timing assumptions are often unrealistic, researchers began to investigate the impact on distributed computing if those timing assumptions are relaxed or tightened to some extent in order to reflect the real time situation. This question has been studied for a variety of problems, including Byzantine agreement [7, 8, 1, 13], mutual exclusion [3], leader election [5], transaction commit [6], and the session problem [2, 4].

The (s, n)-*session* problem, first presented in [2], is an abstraction of the synchronization needed to solve many distributed computing problems. Therefore, it is an important tool for understanding the behavior of distributed systems under different timing constraints. Informally, a *session* is a minimal-length computation fragment that involves at least one "synchronization" step by every process in a distinguished set of n processes. An algorithm that solves the (s, n)-session problem must guarantee that in every computation there are at least s disjoint sessions and eventually all the n processes become idle.

We study the problem in two different interprocess communication models: *shared memory* and *message passing*. In the shared memory model, processes communicate only by means of shared variables. Each variable is shared by no more than b processes, where b is a constant relative to the total number of processes. In the message passing model, communication is done by exchanging messages across a network. A process can broadcast a message at a step; the message is guaranteed to be delivered to every process after some finite time.

The relevant timing aspects of a model are the

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0191...$1.50

lower bound on process step time, c_1, the upper bound on process step time, c_2, and additionally, for the message passing model, the lower bound on message delay, d_1, and the upper bound on message delay, d_2. The running time of an algorithm for the (s, n)-session problem is the maximum time, over all computations, until all the n processes become idle. If there is an upper bound in real time on c_2 and d_2, then it makes sense to measure the running time in terms of real time. If not, then the common way to measure the running time is with rounds. A *round* is a minimal computation fragment in which every process takes at least one step.

Arjomandi, Fischer and Lynch [2] studied the (s, n)-session problem in synchronous and asynchronous shared memory models. *Synchronous* means that $c_1 = c_2$, a finite number. *Asynchronous* means that c_1 and c_2 are infinite. Their results showed a significant time complexity gap between the synchronous and asynchronous models, namely that s rounds are sufficient for the synchronous case but $(s-1)\lfloor \log_b n \rfloor$ rounds are necessary for the asynchronous case, where n is the size of the distinguished set of processes. The implication is that no communication is needed at all in the synchronous case, but it is needed for every session in the asynchronous case. (The $\lfloor \log_b n \rfloor$ factor is essentially the cost of communication when no more than b processes can access any shared variable.)

Attiya and Mavronicolas [4] addressed the problem in semi-synchronous and asynchronous message passing systems. *Semi-synchronous* means that $c_1 > 0$, c_2 and d_2 are finite, and these constants are known to the processes. They modeled the asynchronous system differently than [2]: they let $c_1 = 0$ and $d_1 = 0$, while c_2 and d_2 are finite. Their results also indicated an important time separation between semi-synchronous and asynchronous networks, again based on whether or not communication is necessary.

We present almost matching upper and lower bounds for the session problem in the semi-synchronous shared memory model. Our bounds are similar to those in [4] for the message passing model when the cost for information propagation in the shared memory model is substituted for the message delay. They indicate that if the time for one communication is less than that for one step multiplied by the ratio of c_2 and c_1, the model behaves like the asynchronous; otherwise it behaves like the synchronous (inflated by the ratio). Mavronicolas [12] has also independently developed the same lower and upper bounds for the shared memory semi-synchronous model.

We introduce two new timing models for the (s, n)-session problem: the periodic and the sporadic. In the *periodic* model, for each process there exists an unknown constant such that the process makes one step at every period of the constant. In the message-passing variant, d_2 is finite and known. The upper bounds for both the shared memory and message passing models are the time for the slowest process to take s steps plus the time for one communication. The lower bounds for both are the maximum of the time for the slowest process to take s steps and approximately the time for one communication. Our results indicate that the periodic model, which requires one communication, falls in between the synchronous and asynchronous models, which require no and $s-1$ communications respectively.

In the *sporadic* model, there exists a nonzero lower bound c_1, but no upper bound, on the time between any two consecutive steps of any process. The sporadic shared memory model is essentially equal to the asynchronous shared memory model and is not considered. For the message passing model, the message delay is within $[d_1, d_2]$, where $d_1 \geq 0$, d_2 is finite, and both are known. The combination of the lower bound on step time and upper bound on message delay allows processes to make inferences about the computation, namely, that enough time has elapsed so that a message must have arrived. The lower bound on the per-session time is $\max\{\lfloor \frac{u}{4c_1} \rfloor \cdot K, c_1\}$, where $u = d_2 - d_1$ and $K = \frac{2d_2c_1}{d_2 - u/2}$. The upper bound on the per-session time is $\min\{(\lfloor \frac{u}{c_1} \rfloor + 3) \cdot \gamma + u, d_2 + \gamma\}$, where γ is the largest step time by a process before termination. As the message delay approaches a constant (i.e., d_1 approaches d_2), the per-session time becomes $\max\{0, c_1\} = c_1$ for the lower bound and $\min\{3\gamma, d_2 + \gamma\} = O(\gamma)$ for the upper bound, which is like the synchronous model.

As the message delay fluctuates within a bigger interval (i.e., d_1 approaches 0), the per-session time becomes $\max\{d_2, c_1\} = d_2$ for the lower bound and $\min\{(\lfloor \frac{d_2}{c_1} \rfloor + 3) \cdot \gamma + d_2, d_2 + \gamma\} = O(d_2 + \gamma)$ for the upper bound, which is like the asynchronous model.

These two timing constraints are inspired by constraints with the same names commonly used in

many real-time problems, especially in scheduling of real time tasks for a uniprocessor [9, 10, 11] where the period of task occurrences conforms to the constraints. In practice, as quoted in [10], periodic timing constraints are used in applications such as avionics and process control when accurate control requires continual sampling and processing of data. The sporadic timing constraint is associated with event-driven processing such as responding to user inputs or non-periodic device interrupts; these events occur repeatedly, but the time interval between consecutive occurrences varies and can be arbitrarily large. Therefore, the sporadic timing constraint models processes that can be blocked for an arbitrarily long (but finite) time waiting for a certain condition to be true or a certain event to occur, but that cannot make two consecutive steps faster than a certain lower bound.

Table 1 summarizes the bounds. L means lower bound, U means upper. In the periodic model, c_{min} and c_{max} are the smallest and the largest step times respectively of all processes. The bounds for the asynchronous shared memory case are in rounds. The bounds from [4] have been converted in three aspects for purposes of comparison: (1) That paper considers point-to-point networks; thus the results include a factor of the network diameter. In our model, d_2 subsumes the diameter factor; we have replaced all occurrences of the diameter factor with 1. (2) In [4], the constant 1 is used as the value of c_2; we have replaced all appropriate occurrences of 1 with c_2. (3) [4] assumes that all processes take their synchronized first steps at time 0, resulting in one session at time 0; although we assume that all processes start at time 0, we don't assume that all take a synchronized step at time 0. We rather assume that all steps (including the first step) obey the timing constraints of a specific model starting time 0.

Our results indicate that the periodic model is more efficient than the semi-synchronous system when $c_{max} = c_2$, $2c_1 < c_2$ and n is constant relative to s. The lower bound for the sporadic system and the upper bound for the periodic system suggest that the periodic system is more efficient than the sporadic system if c_{max} is smaller than $\lfloor \frac{u}{4c_1} \rfloor \cdot K \leq \frac{d_2 u}{2(d_2 - u/2)}$ and n is constant relative to s. In shared memory, the sporadic system is clearly less efficient than the semi-synchronous one, but the relationship between the sporadic and the semi-synchronous systems for message passing is rather unclear and understanding it requires further study.

The rest of the paper is organized as follows. Section 2 contains the definition of system models and Section 3 describes how to accomplish communication in the shared memory model. Section 4 concerns the periodic model, Section 5 the shared memory semi-synchronous, and Section 6 the message-passing sporadic. Please note that our lower bound proof technique combines those in [2, 4]. Some proofs are omitted or only sketched due to space constraints.

2 Definitions

2.1 Systems

The generalized system model definition for shared memory and message passing models is similar to that defined in [2].

There are finite sets P of processes and X of shared variables. A *process* consists of a set of internal states, including an initial state. Each *shared variable* has a set of values that it can take on, including an initial value. A *global state* is a tuple of internal states, one for each process, and values, one for each shared variable. The *initial* global state contains the initial state for each process and the initial value for each shared variable. A *step* π consists of simultaneous changes to the state of some process and the values of some number of variables, depending on the current state of that process and current values of the variables. More formally, we represent the step π with a tuple $((s, p, r), (u_1, x_1, v_1), \ldots (u_k, x_k, v_k))$, where s and r are old and new states of a process $p \in P$; u_i and v_i are old and new values of a shared variable $x_i \in X$ for all i. We say that step π is *applicable* to a global state if p is in state s and x_i has value u_i for all i in the global state.

A *system* is specified by describing P, X, and set Σ of possible steps. For all processes $p \in P$ and all global states g, there must exist some step involving process p that is applicable to global state g. A *computation* of a system is a sequence of steps $\pi_1, \pi_2, \ldots$ such that: (1) π_1 is applicable to the initial global state, (2) each subsequent step is applicable to the global state resulting from the previous steps, and (3) if the sequence is infinite, then every process takes an infinite number of steps. That is, there is no process failure. A *timed computation* of

Model		Shared Memory	Message Passing
Sync.	L	$s \cdot c_2$ [2]	$s \cdot c_2$
	U	$s \cdot c_2$ [2]	$s \cdot c_2$
Peri-odic	L	$\max\{s \cdot c_{max}, \lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}\}$	$\max\{s \cdot c_{max}, d_2\}$
	U	$s \cdot c_{max} + O(\log_b n) \cdot c_{max}$	$s \cdot c_{max} + d_2$
Semi-sync.	L	$\min\{\lfloor \frac{c_2}{2c_1} \rfloor \cdot c_2, \lfloor \log_b n \rfloor \cdot c_2\} \cdot (s-1)$	$\min\{\lfloor \frac{c_2}{2c_1} \rfloor \cdot c_2, d_2 + c_2\} \cdot (s-1)$ [4]
	U	$\min\{(\lfloor \frac{c_2}{c_1} \rfloor + 1) \cdot c_2, O(\log_b n) \cdot c_2\} \cdot (s-1) + c_2$	$\min\{(\lfloor \frac{c_2}{c_1} \rfloor + 1) \cdot c_2, d_2 + c_2\} \cdot (s-1) + c_2$ [4]
Spor-adic	L	See Async. SM	$\max\{\lfloor \frac{u}{4c_1} \rfloor \cdot K, c_1\} \cdot (s-1)$
	U	See Async. SM	$\min\{(\lfloor \frac{u}{c_1} \rfloor + 3) \cdot \gamma + u, d_2 + \gamma\} \cdot (s-1) + \gamma$
Async.	L	$(s-1) \cdot \lfloor \log_b n \rfloor$ [2]	$(s-1) \cdot d_2$ [4]
	U	$(s-1) \cdot O(\log_b n)$ [2]	$(s-1) \cdot (d_2 + c_2) + c_2$ [4]

Table 1: Bounds for the Session Problem

a system is a computation $\pi_1, \pi_2, \ldots$ together with a mapping T from positive integers to real numbers that associates a real time with each step in the computation. T must be nondecreasing and, if the computation is infinite, increase without bound. We will abuse notation and let $T(\pi_i)$ indicate the time at which step π_i occurs.

2.1.1 Shared Memory Model (SMM)

We specialize the general system into the shared memory system in which processes communicate with each other by means of shared variables. Each step π has the property that $k = 1$. That is, it involves only one shared variable. A process can read and write a shared variable in a single atomic step, and we don't assume any upper bound on the size of the variables. We let b be the maximum number of processes that access any single variable, in all the steps of the system. We assume b is constant relative to the number of processes.

2.1.2 Message Passing Model (MPM)

We specialize the general system into the message passing system, in which processes communicate with each other by exchanging messages. $P = R \cup \{N\}$, where R is the set of *regular* processes and N is the network. The network schedules the delivery of messages sent among the regular processes. $X = \{net\} \cup \{buf_p : p \in R\}$, where the values taken on by each variable are sets of messages. *net* models the state of the network, i.e., the set of messages in transit. buf_p holds the set of messages that have been delivered to p by the network but not yet received by p.

A step of a process p in R consists of p receiving the set M of messages in its buffer buf_p, and based solely on those message and its current state, changing its local state and sending out some message m to all the regular processes. The result is to set buf_p to empty and to add (m, q) to *net*, for all q in R. So, the step involves two shared variables, buf_p and *net*. A step of N is to deliver some message of the form (m, q) in *net* to q. The result is to remove (m, q) from *net* and add m to buf_q. Accordingly, the step also involves two shared variables, *net* and buf_q.

This definition of the MPM is an abstract model of a reliable strongly connected network with any topology.

In a timed computation, each message has a *delay*, defined to be the difference between the time of the step that adds it to *net* and the time of the step that removes it from *net*. If the message is never removed, then it has infinite delay. The delay only counts the time in transit in the network and does not include the time that the recipient takes to receive the message. That is, the time elapsed between the delivery step and the step which finally removes the message from the buffer is not counted toward the message delay, even if the message remains in the buffer for a long time before the recipient picks it up from its buffer.

2.2 The Real Time Constraints

For each timing model considered, we define the set of *admissible* timed computations to consist of timed computations which obey the stated condition on the step times of all processes in the SMM

(all regular processes in the MPM) and, additionally for the MPM, the stated condition on the message delay.

Synchronous There exist constants c_2 and d_2 such that in every timed computation, for every p in P (p in R for MPM), the time between every pair of consecutive steps of p is c_2, and the delay of every message is d_2. Thus c_2 and d_2 are "known" to the processes and can be used in algorithms.

Asynchronous In every timed computation, every process takes an infinite number of steps and every message is eventually delivered.

Periodic There exists a constant d_2 such that in every timed computation, for every p_i in P (p_i in R for MPM), there exists constant c_i such that the time between every pair of consecutive steps of p_i is c_i, and the delay of every message is in $[0, d_2]$. Thus the c_i's are unknown but d_2 is known.

Semi-Synchronous There exist constants $c_1 > 0$, c_2 and d_2 such that in every timed computation, for every p in P (p in R for MPM), the time between every pair of consecutive steps of p is in $[c_1, c_2]$ and the delay of every message is in $[0, d_2]$. Thus c_1, c_2 and d_2 are known.

Sporadic There exist constants c_1, d_1, and d_2 such that in every timed computation, for every p in P (p in R for MPM), the time between every pair of consecutive steps of p is at least c_1, and the delay of every message is in $[d_1, d_2]$. Thus c_1, d_1, and d_2 are known.

2.3 The Session Problem

We now state the conditions that must be satisfied for a system to *solve the* (s, n)*-session problem* (also called an (s, n)-session algorithm).

(1) Each process in P (in R for the MPM) has a subset of *idle* states. The set Σ of steps of the system guarantees that once a process is in an idle state, it always remains in an idle state.

(2) There is a distinguished set Y of n variables called *ports*; Y is a subset of X in the SMM and the set of *buf*'s in the MPM. There is a unique process in P (in R for the MPM) corresponding to each port, which is called a *port process*.

(3) Let p be a port process which corresponds to a port y. A *port step* is any step involving p and y. A *session* is any minimal sequence of steps containing at least one port step for each port in Y. In every admissible timed computation, there are at least s disjoint sessions and eventually all port processes are in idle states.

In the timing models with finite upper bounds on step time and message delay, we measure the running time of an algorithm in real time as follows. An algorithm *runs in time* t if, for every admissible timed computation, every process is in an idle state by time t. In the case of the asynchronous and sporadic models, step time and/or message delay is unbounded (but finite). For these cases, we measure the running time in rounds [14, 2, 4]. A *round* is a minimal-length computation fragment in which every process appears at least once. An algorithm *runs in* r *rounds* if, in every admissible timed computation C, the prefix of C before all processes are idle consists of at most r disjoint rounds. It is also informative in these models to express the time complexity of an algorithm in terms of a new parameter γ, the largest step time during the computation of the algorithm before all the processes are idle. The values of γ is dependent on a particular computation of the algorithm. This type of per-computation based time complexity measure is also used in [1].

3 Communication in SMM

In describing our algorithms, we use a subroutine called *broadcast* as a generic operator for communication in both of the communication models.

In the MPM, the broadcasting of message m by process p is taken care of by the network. It takes at most $d_2 + c_2$ time for a message to be received by all processes in the MPM.

However, the communication in the SMM is constrained by the number of processes which can access a shared variable. Therefore, broadcasting in the SMM involves relaying messages from process to process by means of shared variables.

In a *b-bounded* shared memory system, we can build a tree networks of processes and shared variables by making port and port processes the leaves of the tree. This network can accomplish the necessary communication to propagate a peice of information originaing from a process to all other processes in $O(\log_b n)$ steps.

In this paper, when we say *broadcast* in the SMM, it implies all the interaction of processes in the tree

network to accomplish the broadcasting. Throughout this paper, we only describe the role of port processes in an algorithm and assume that broadcast encapsulates the interactions among port processes and other processes which participate in the tree-network communication. In addition, we use the term "step" interchangeably with "port step"; when necessary, we make the proper distinction.

4 The Periodic Model

The periodic model and the synchronous model are similar in that a process takes steps at regular time intervals, yet they differ from each other in that there is no bound on the relative speed of processes in the periodic model. We first present an algorithm $A(p)$ for the (s,n)-session problem in this model and then show that for all periodic algorithms which solve the (s,n)-session problem, there exists a computation of A which takes at least $\max\{s \cdot c_{max}, d_2\}$ time for the MPM and $\max\{s, \lfloor \log_b n \rfloor\} \cdot c_{max}$ for the SMM.

Algorithm $A(p)$: (This algorithm runs in the MBM and the SMM.) Each port process accesses its own port $s-1$ times and at its $s-1$th step, broadcasts the fact. It enters an idle state after it hears that all other processes have taken $s-1$ steps and it has taken at least one more port step.

Theorem 4.1 *$A(p)$ solves the (s,n)-session problem in time $s \cdot c_{max} + d$ in the MPM and time $s \cdot c_{max} + O(\log_b n) \cdot c_{max}$ in the SMM, where $c_{max} = \max\{c_i : p_i \in P\}$.*

Theorem 4.2 *No MP periodic algorithm for the (s,n)-session problem runs in time less than $\max\{s \cdot c_{max}, d_2\}$.*

Theorem 4.3 *No SM periodic algorithm for the (s,n)-session problem runs in time less than $max\{s \cdot c_{max}, \lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}\}$.*

Proof: Suppose that $s \cdot c_{max} \geq \lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}$. Since all processes must take at least s steps to have s sessions, $s \cdot c_{max}$ is obviously the lower bound.

Suppose that $s \cdot c_{max} < \lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}$. By way of contradiction we assume that there exists an algorithm A which solves the (s,n)-session problem in the periodic SMM in time Z strictly less than $\lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}$. We prove that there exists an infinite admissible computation of A that contains less than s sessions.

Let (α, T) be the admissible timed computation in which processes take steps in round robin order and each process's ith step occurs at time $i \cdot c_{min}$ Each consecutive group of steps for p_1 through $p_{|P|}$ is a round. (Round i occurs at time $i \cdot c_{min}$ and consists of the i th step of each process.) Since all processes should enter idle states by time Z in α and all the step time periods are equal to c_{min} in (α, T), there are at most $r = \lfloor Z/c_{min} \rfloor$ rounds required until termination in α.

We will perturb (α, T) in order to get a new admissible timed computation (α', T'). We will prove that there exists at least one port process in (α', T') which enters an idle state before another port process takes any step, resulting in an admissible computation that contains less than s sessions.

Fix any port process p' and change p's step time period to be $\lfloor \log_{2b-1}(2n-1) \rfloor \cdot c_{min}$. Run A with this modified set of processes to get a new timed admissible computation (α', T').

We define a *subround* to be a minimal computation fragment of α' that involves all processes except p'. A variable v is *contaminated* in subround k of α' if there exists $j \leq k$ and process $p \neq p'$ such that v's value in the global state of α' following p's step on subround j is not equal to v's value in the global state of α following p's step in round j. We define no variable to be contaminated in subround 0. A process p is *contaminated* in subround k of α' if $p \neq p'$ and there exists $j \leq k$ such that in subround j of α', p accesses a variable that is contaminated in subround j. We define no processes to be contaminated in subround 0.

Let $P(t)$ be the set of processes that are contaminated in subround t, and let $V(t)$ be the set of variables that are not contaminated in subround $t-1$ but are contaminated in subround t. Let P_t and V_t satisfy the recurrence equations: $P_0 = V_0 = 0$, $V_t = 2 \cdot P_{t-1} + 1$, and $P_t = (b-1) \cdot V_t + P_{t-1}$.

Lemma 4.4 *$|P(t)| \leq P_t$ and $|V(t)| \leq V_t$ for $0 \leq t \leq r$, where $r = \lfloor Z/c_{min} \rfloor$.*

Proof: By induction on t. The key points are that p' contributes at most one variable to $V(t)$, while each contaminated process contributes at

most two. Also, in the worst case a process becomes contaminated as soon as possible, processes only become contaminated due to the variables that just become contaminated, and each variable contaminates at most $b-1$ other processes. ■

Soving the recurrence equation, we get

$$P_t = \frac{(2b-1)^t - 1}{2}.$$

Thus the total number of processes that are contaminated in subround r is at most $n-1$.

Since less than n processes are contaminated in subround r, at least one port process $p \neq p'$ is in the same state at the end of subround r in α' as it is at the end of round r in α — an idle state. But p' has not taken a step yet. Thus (α', T') is an admissible timed computation that contains less than s sessions. Contradiction. (Note that $\log_{2b-1}(2n-1)$ approaches $\log_b n$ as b and n increase.) ■

5 Semi-Synchronous Model

In this section, we address the upper and lower bounds in the semi-synchronous shared memory model. The semi-synchronous algorithm in [4] can be adapted to work in the shared memory semi-synchronous model simply by replacing the communication primitives (send and receive) with the explicit propagation of information through the tree network of shared variables using the *broadcast* subroutine described in Section 2.

The proof of the lower bound for the semi-synchronous SMM is rather complicated, because the propagation of information relies on reading and writing shared variables, and also because computations constructed in the proof must satisfy the real time constraints.

Theorem 5.1 *There is no semi-synchronous algorithm which solves the (s,n)-session problem in the SMM within time strictly less than* $\min\{\lfloor \frac{c_2}{2c_1} \rfloor, \lfloor \log_b n \rfloor\} \cdot c_2 \cdot (s-1)$.

Proof: Let $B = \min\{\lfloor \frac{c_2}{2c_1} \rfloor, \lfloor \log_b n \rfloor\}$.

If $c_2 \leq 2c_1$, then $B \leq 1$ and it is obvious that the bound holds since every process must take at least s steps to have s sessions.

Suppose $c_2 > 2c_1$. Assume, by way of contradiction, that there exists a semi-synchronous algorithm, A, which solves the problem in SMM within time Z strictly less than $B \cdot c_2 \cdot (s-1)$. Then $\lceil Z/(B \cdot c_2) \rceil \leq (s-1)$.

Let (α, T) be the admissible timed computation in which processes take steps in round robin order and each process' ith step occurs at time $i \cdot c_2$. Each consecutive group of steps for p_1 through $p_{|P|}$ is a round.

There are $t = \lceil Z/c_2 \rceil$ rounds required until termination in α. Let $\alpha = \beta\gamma$, where β contains the first t rounds of α.

Following the proof of Theorem 1 in [2], we will show that there is a reordering β' of β that results in the same global state as β but that contains at *most $s-1$ sessions*. Thus $\alpha' = \beta'\gamma$ is a computation with at most $s-1$ sessions. We then will show how to time the events in α' to produce an admissible timed computation (α', T') with at most $s-1$ sessions, a contradiction.

We construct a partial order $\leq_\beta$ on the steps in β, representing dependency. Let $\sigma \leq_\beta \tau$ for every pair of steps σ and τ in β, and say τ is *dependent* on σ, if $\sigma = \tau$ or if σ precedes τ in α and σ and τ either involve the same process or involve the same variable. Close $\leq_\beta$ under transitivity. The following claim is not difficult to prove.

Claim 5.2 *$\leq_\beta$ is a partial order, and every total order of steps of β consistent with $\leq_\beta$ is a computation which leaves the system in the same global state as β does.*

Let $\beta = \beta_0, \beta_1, \ldots, \beta_m$ where $m = \lceil Z/(B \cdot c_2) \rceil$. Each β_k (except possibly the last one) consists of B rounds. Let y_0 be an arbitrary port in Y. For all k, $1 \leq k \leq s-1$, we show that there exists a port y_k and two sequences of steps ϕ_k and ψ_k, such that the following properties hold. (p_{y_i} is the corresponding port process to y_i, $1 \leq i \leq s-1$.)

(i) $\phi_k\psi_k$ is a total ordering of the steps in β_k, consistent with $\leq_\beta$.

(ii)ϕ_k does not contain any step by process $p_{y_{k-1}}$ which accesses y_{k-1}.

(iii)ψ_k does not contain any step by process p_{y_k} which accesses y_k.

Then define β' to be $\phi_1\psi_1\phi_2\psi_2 \ldots \phi_m\psi_m$.

For all k, $1 \leq k \leq m$, y_k, ϕ_k, and ψ_k are defined inductively. If there is some port variable that is not accessed by any step in β_k, then let y_k be that port, ϕ_k the empty sequence, and $\psi_k = \beta_k$. Otherwise (every port variable is accessed in β_k), let τ_k be the first step in β_k that accesses y_{k-1}. As a consequence of a very general result proved in [1], there exists a port variable y_k such that:

(iv) it is not the case that $\tau_k \leq_\beta \sigma_k$, where σ_k is the last step in β_k that accesses y_k.

We now assign times (the mapping T') to every step in β_k and then let β'_k be any total ordering of the steps in β_k consistent with the times. We then define ϕ_k and ψ_k.

- For each process $p \in P$, if there are some steps of p in β_k which σ_k is dependent on, we let π be the step that occurred last among them. We retime π and all the steps of p that happened earlier than π such that the first step of p in β_k occurs at $2c_1B(k-1)+c_1$, the next step occurs c_1 time later, and so on.
- For each process p, if there are some steps of p in β_k which are dependent on τ_k, we let π be the step that occurred first among them. We retime π and all the steps of p that occurred later than π such that the last step of p in β_k occurs at $2c_1Bk$, the step before that occurs c_1 time earlier, and so on.
- All other steps in β_k are assigned the same time as they are under T (the original timing).

Let ϕ_k be all the steps that happened up to $time(\sigma_k)$ including σ_k, and let ψ_k be the remainder.

Lemma 5.3 *β' is consistent with $\leq_\beta$.*

Proof: For any k, $1 \leq k \leq s-1$, pick any two steps, π and π' in β_k such that $\pi \leq_\beta \pi'$. Thus $T(\pi) \leq T(\pi')$. (Recall that T is the original timing.) We only need to prove that $T'(\pi) \leq T'(\pi')$, where T' is the new timing.

Each of π and π' belongs to either ϕ_k or ψ_k and is either retimed or not. If π is retimed in ϕ_k and π' is not retimed in ϕ_k, then $T'(\pi) \leq T'(\pi')$ since $T'(\pi) \leq T(\pi)$ and $T'(\pi') \geq T(\pi')$. All other cases can be proved similarly. ■

Lemma 5.4 *(β', T') is admissible.*

Proof: We need to prove that all steps in (β', T') satisfy the real time constraint imposed on the semi-synchronous model.

By the construction, no two consecutive steps by a process in the system are closer than c_1 in (β', T'); therefore, the lower bound on step time is preserved.

We now show that the maximum time between any two consecutive steps of a process is c_2. Let π and π' be two consecutive steps of some process p. First assume that π and π' both occur in β_k for some k, π is the i-th step of p in β_k and π' is the $i+1$-st. If π and π' are both in ϕ_k or are both in ψ_k, then either there is no change in their times or they are retimed to be c_1 apart.

Now suppose π is in ϕ_k and π' is in ψ_k. By construction,

$$\begin{aligned} & T'(\pi') - T'(\pi) \\ &\approx B \cdot c_1 + c_1 \\ &\approx \min\{\lfloor \frac{c_2}{2c_1} \rfloor, \log_b n\} \cdot c_1 + c_1 \\ &\leq \frac{c_2}{2} + c_1. \end{aligned}$$

Since $c_1 < c_2/2$, this difference is less than c_2.

Now suppose π occurs in β_{k-1} and π' occurs in β_k. In the worst case, π is retimed to occur at $x - c_2/2$ and π' is retimed to occur at $x + c_2/2$, where x is the time at the end of β_{k-1}. (This is true by the definition of B.) So the time between π and π' is at most c_2. ■

Lemma 5.5 *β' contains less than s sessions.*

Lemma 5.5 is true because of the way ψ_{k-1} and ϕ_k are defined. The theorem now follows. ■

6 The Sporadic Model

In the MPM, a lower bound c_1 on step time and lower and upper bounds $[d_1, d_2]$ on message delay are imposed. The correctness of our sporadic algorithm $A(sp)$ depends on the following observation: If a process p_i receives a message m from a process p_j at time t, then the message must have been sent no later than $t-d_1$, because it takes at least d_1 time for a message to be delivered. All the messages received by p_i after $t+d_2-d_1$ must have been sent

after m was, because it takes at most d_2 time for a message to be delivered.

Using the above fact, each process broadcasts a message at every step carrying its knowledge on the number of sesssions happened by the time that the step occurs. After receiving a message m which says there are at least $k-1$ sessions in the system, a process waits for $d_2 - d_1$ time. After that, the process waits to receive at least a message from every process. it is clear that there are at least k sessions in the system by the time because every message received after $t + d_2 - d_1$, where t is the time that m is sent, must have been sent after the time there are at least $k-1$ sessions.

We first proceed by presenting the algorithm $A(sp)$. A message is denoted $m(i, V)$, where i is the identifier of a sending process p_i and V is an integer in $[0, s-1]$. We let $*$ be a *don't care* value for either of the fields and $u = d_2 - d_1$.

$A(sp)$ for process p_i:
$B := \lfloor \frac{u}{c_1} \rfloor + 1$;
count := *session* := 0;
msg_buf := *temp_buf* := $\emptyset$;
while(*session* $< s-1$)
  read buf_i and let the set of messages
    obtained be M;
  msg_buf := *msg_buf* $\cup M$;
  if for all $j \in [n]$, $m(j, session)$ is in *msg_buf*
  then /* condition 1 */
    count := 0;
    session := *session* + 1;
  elsif (*count* $> B$)
  then
    temp_buf := *temp_buf* $\cup M$;
    if for all $j \in [n]$, at least one $m(j, *)$
      is in *temp_buf*
    then /* condition 2 */
      count := 0;
      session := *session* + 1;
      temp_buf := $\emptyset$;
    end if;
  end if;
  broadcast $m(i, session)$;
  count := *count* + 1;
end while;
Enter an idle state.

Theorem 6.1 *$A(sp)$ solves the (s, n)-session problem within time*
$\min\{\lfloor \frac{u}{c_1} + 1 \rfloor \gamma + (u + 2\gamma), d_2 + \gamma\}(s-2) + d_2 + 2\gamma$.

Proof: Consider an arbitrary admissible timed computation C of $A(sp)$. The following lemma (proof omitted) can be used to prove the theorem.

Lemma 6.2 *There exists at least one step in C in which a process sets its session to k, $0 \le k \le s-1$.*

Let p_{i_k} be the first process which sets $session_{i_k}$ to $k \ge 0$. To increment *session*, a process must receive a message(s) which notifies the process that there is at least one session after the last update to *session*. Let M_k be the set of messages received by p_{i_k} that causes p_{i_k} to set $session_{i_k}$ to k. (We define M_0 to be the empty set.) In more detail: If condition 1 was true, M_k is the set of message $m(j, session_{i_{k-1}})$ for all integers $j \in [n]$ in *msg_buf*. If condition 2 was true, M_k is the set of messages in *temp_buf* at the time. Assuming that m_k is the message which is sent last among M_k, we prove the following lemma.

Lemma 6.3 *Let π be the step which sends m_k. There are at least k sessions by the time π occurs in C.*

Proof: We proceed by induction on k.

For the basis, when $k = 0$, it is always true that there are at least 0 sessions in C.

Inductively when $k > 0$, assuming the lemma is true for $k-1$, we show that when π occurs, there are at least k sessions.

Let τ be the step that sent m_{k-1} and σ be the step in which $p_{i_{k-1}}$ sets $session_{i_{k-1}}$ to $k-1$. For p_{i_k} to update its *session*, one of conditions 1 and 2 in the algorithm must hold.

First, assume that condition 1 is true. According to the algorithm, a process broadcasts a message with a new session value $k-1$ after it sets its *session* to the new value $k-1$, before which time there were $k-1$ sessions in C because the induction hypothesis dictates that there were $k-1$ sessions in C when m_{k-1} was sent. Because σ is the first step to set $session_{i_{k-1}}$ to $k-1$, all messages in M_k, must have been sent when or after σ occurs. Because all processes take at least one step to send messages in M_k after there were at least $k-1$ sessions, there must be at least k sessions in C by the time that message m_k is sent.

Second, assume that condition 2 is true. Since p_{i_k} is the first process which sets $session_{i_k}$ to k,

$session_{i_k}$ must have taken on $k-1$ according to the proof of Lemma 6.2. Let t be the time when τ occurs at which time m_{k-1} was sent and t' be the time that p_{i_k} sets $session_{i_k}$ to $k-1$. The message m_{k-1} must arrive at $buf_{i_{k-1}}$ at time between $[t+d_1, t+d_2]$ because of the bounds on message delay. Thus, $t'-t \geq d_1$. Since *count* in the algorithm is reset whenever *session* is updated, when $count_{i_k}$ is equal to B most recently before when p_{i_k} sets $session_{i_k}$ to k, say, at time t'', at least time $B \cdot c_1 > u$ must have elapsed since t'. So, $t'' > t' + u \geq t + d$. Therefore, all messages received at t'' or later must be sent after time t, at which time there were $k-1$ sessions by the assumption. Since at least one message is sent by each process after time t, there must be at least one additional step by all processes between time t and the time π occurs. Therefore, there must be at least k sessions by the time π occurs. ■

To analyze the time complexity of the algorithm $A(sp)$, we use the actual maximum step time γ since in our sporadic model the upper bound on the step time is not available.

We define for each k, $2 \leq k \leq s-1$, $T_k = \max\{t : p_i \text{ sets } session_i \text{ to } k \text{ at time } t \text{ in } C \text{ for all } p_i \in R\}$.

Lemma 6.4 *For each k, $2 \leq k \leq s-1$, $T_{k+1} \leq T_k + \min\{(\lfloor \frac{u}{c_1} + 1 \rfloor)\gamma + (u+2\gamma), d_2 + \gamma\}$.*

Proof: According to the algorithm, a process broadcasts a message at every step. Thus, if process p_i receives a message from process p_j at time t, it will receive at least one more message from p_j by time $\leq t + u + 2\gamma$. Let p_{i_k} be the last process to set $session_{i_k}$ to k and $p_{i_{k+1}}$ be the last process to set $session_{i_k}$ to $k+1$. We now look at each of the possible cases which may cause $session_{i_{k+1}}$ to be updated to $k+1$:

If condition 2 is true when $session_{i_{k+1}}$ is updated to $k+1$, $p_{i_{k+1}}$ has made at least $B = \lfloor \frac{u}{c} + 1 \rfloor$ steps since the last update to $session_{i_{k+1}}$. Because a process must wait, since then, at most $u + 2\gamma$ time to receive another set of messages sent from all processes, at most $(\lfloor \frac{u}{c} + 1 \rfloor)\gamma + (u + 2\gamma)$ time has elapsed.

If condition 1 is true when $session_{i_{k+1}}$ is updated to $k+1$, let t be the time at which p_{i_k} broadcasts $m(i_k, k)$; note that by definition, $t = T_k$. Message $m(i,k)$ must be received by $p_{i_{k+1}}$ by time $t + d_2 + \gamma$.

Since both conditions take at most $\min\{\lfloor \frac{u}{c} + 1 \rfloor \gamma + (u + 2\gamma), d_2 + \gamma\}$ time to be true since the last update to *session*, the lemma follows. ■

From Lemmas 6.2 and 6.3, it follows that there are at least $s-1$ sessions at the time that m_{s-1} is sent. All processes will eventually set their *session*'s to $s-1$ (since *session* can't be bigger than $s-1$). Each process sets *session* to $s-1$ because it receives a certain message. Therefore, there is at least one additional step by all processes after there have been $s-1$ sessions in C. Thus, there are at least s sessions in C.

By the algorithm, initially it takes at most $d_2 + 2\gamma$ to receive at least one message from all processes in order to accomplish the first session. Using Lemma 6.4, it is clear now that it takes at most $\min\{\lfloor \frac{u}{c_1} + 1 \rfloor \gamma + (u + 2\gamma), d_2 + \gamma\}(s-2) + d_2 + 2\gamma$. (This equals $\min\{(\lfloor \frac{u}{c_1} \rfloor + 3) \cdot \gamma + u, d_2 + \gamma\}(s-1) + \gamma$ if $d_1 < \lfloor \frac{u}{c_1} + 1 \rfloor \cdot \gamma$). ■

We now prove the lower bound.

Theorem 6.5 *No sporadic algorithm solves the (s,n)-session problem in the MPM within time $< \max\{\lfloor \frac{u}{4c_1} \rfloor \cdot K, c_1\}(s-1)$ where $K = \frac{2d_2c_1}{(d_2 - \frac{u}{2})}$.*

Proof: The general structure of this proof follows that of Theorem 5.1.

Let $B = \lfloor \frac{u}{4c_1} \rfloor$.

When $B \cdot K \leq c_1$, the lower bound holds because a process must execute at least s steps to achieve s sessions.

Suppose that $B \cdot K > c_1$. Assume, by way of contradiction, that there exists a sporadic algorithm, A, which solves the (s,n)-session problem in the MPM within time Z strictly less than $B \cdot K \cdot (s-1)$. Then $\lceil Z/(B \cdot K) \rceil \leq (s-1)$. We show that there exists an admissible timed computation of A which does not include s sessions.

Let (α, T) be the admissible timed computation in which regular processes take steps in round robin order and each process' ith step occurs at time $i \cdot K$, and all message delays are exactly d_2. Each consecutive group of steps for p_1 through p_n is a round.

Therefore, there are $r = \lceil Z/K \rceil$ rounds required until termination in α. Let $\alpha = \beta\gamma$, where β contains the first r rounds of α.

We will show that there is a reordering β' of β that contains at most $s-1$ sessions. Thus $\alpha' = \beta'\gamma$ is a sequence with at most $s-1$ sessions. In order to get an admisssible computation β', we will assign new times (T') to every step in β and let β' be any total ordering of the steps in β consistent with the times, and then we will prove that (β', T') is an admissible timed computation which results in the same global state as β. A contradiction.

Then we will show how to reorder β to produce admissible timed computation (α', T') that results in the same global state as α.

Let $\beta = \beta_1 \ldots \beta_m$ where $m = \lceil Z/(B \cdot K) \rceil$. Each β_k, $1 \leq k \leq m$ (except possible the last one) consists of B rounds, and for some sequence $i_0, i_1 \ldots i_m$ of integers in $[1, n]$, each computation fragment β_k consists of $\phi_k \psi_k$ such that:

(i) ϕ_k does not contain any step by process $p_{i_{k-1}}$.

(ii) ψ_k does not contain any step by process p_{i_k}.

Then define β' to be $\phi_1 \psi_1 \phi_2 \psi_2 \ldots \phi_m \psi_m$. As in Lemma 5.5, we prove that β' has at most $s-1$ session.

Lemma 6.6 *β' has at most $s-1$ sessions.*

Since all processes in γ are in idle states, α' has at most $s-1$ sessions.

We need to show how to reorder every step in β to get an admissible timed computation (α', T') which preserves properties (i) and (ii), and results in the same global state as α.

Let us first assign times (a new mapping T'') to every step, π, in β including all the steps of the network N such that $T''(\pi) = T(\pi) \cdot \frac{2c_1}{K}$. That is, every process except N takes a step at every $2c_1$ and each round occurs at every $2c_1$. Since the delivery steps of N are also remapped, the message delay is reduced to $d_2 \cdot \frac{2c_1}{K} = d_2 - \frac{u}{2}$.

$C' = (\beta, T'')$ is an admissible timed computation because β is a computation, and the step times and message delays obey the sporadic time constraint.

From C', we construct (β', T'), an admissible computation which results in the same global state as β'' (and β). We map T'' to T' in order to get i_k, i_{k-1}, ϕ_k and ψ_k for all k, $1 \leq k \leq m$.

For all k, choose i_k arbitrarily, as long as $i_k \neq i_{k-1}$. For all $0 \leq j \leq m$, let $t_j = B \cdot 2c_1 \cdot j$. t_j is equal to the ending time of β_j in C'.

1. Let π be all steps of p_{i_k} and all the steps of N that deliver messages to p_{i_k} in β_k. Retime π such that $T'(\pi) = t_{k-1} + (T''(\pi) - t_{k-1})/2$.
2. Let σ be all steps of $p_{i_{k-1}}$ and all the steps of N that deliver messages to $p_{i_{k-1}}$ in β_k. Retime σ such that $T'(\sigma) = t_k - (t_k - T''(\sigma))/2$.
3. All other steps in β_k are assigned the same time as they are under T''.

Fix up the states of the network in β so that the state of the network is consistent with all the send and receive steps of regular processes in β. Let ϕ_k be the prefix of β'_k up to the last step of p_{i_k}, and let ψ_k be the remainder. Let β'_k be any total ordering consistent with T'.

We now prove the following lemma:

Lemma 6.7 *(β', T'') is an admissible timed computation which results in the same global state as β.*

Proof: The time period of β_k in C', $1 \leq k \leq m$, is equal to to $B2c_1$ since β_k in C' consists of B rounds (except the last one) and the step time is c_1. Since no step is retimed outside the time boundary of (β_k, T'), the time period of (β_k, T'') is also equal to (β'_k, T'').

In (β, T''), the message delay of all messages is bigger than $B2c_1$ by the definition of B, so that the messages sent in β_k are never received in β_k. In (β', T'), the messages sent in β'_k are never received in β'_k too because no step is retimed outside the the time boundary of (β_k, T'').

By the construction, in β', the delivery steps of all messages are retimed with the steps that receive the messages, so that every message is delivered always before received. Every step in β' receives the same set of messages as the corresponding step in β does. Since states of processes are updated based only on the current state and the set of message received, β' is a computation which leads to the same global state as β.

Now, to prove that (β', T') is admissible, we need to show that (β', T') obeys the sporadic time constraints.

First, it is clear that every computation step time in β' is bigger than the minimum step time c_1 by the construction.

Second, we need to prove the delay of any message sent in β' is within $[d_2-u, d_2]$. For any message m sent in β', let π_i be the step of a process in R which sends m and π_j be the step of N which delivers m. We need to prove that $T'(\pi_j) - T'(\pi_i)$ is in $[d_2-u, d_2]$. Without loss of generality, assuming π_i is in β'_k, $T'(\pi_j) - T'(\pi_i) = T''(\pi_j) - T''(\pi_i) + [T'(\pi_j) - T''(\pi_j)] - [T'(\pi_i) - T''(\pi_i)]$.

It can be proved that for any step π,

$$-u/4 \leq T'(\pi) - T''(\pi) \leq u/4.$$

$T''(\pi_j) - T''(\pi_i) = B \cdot 2c_1 \leq d_2 - u/2$ from the construction of C'. Therefore, it is clear that $T'(\pi_j) - T'(\pi_i)$ is always within $[d_2 - u, d_2]$ in β'. ■

Since there exists an admissible timed computation (β', T') of A which has at most $s-1$ sessions by Lemma 6.7 and Lemma 6.6, this contradicts the assumed existence of algorithm A. Therefore, Theorem 6.5 now follows. ■

Acknowledgments:

We thank Kevin Jeffay for helpful comments. This work was partially supported by an NSF PYI Award CCR-9158478, an IBM Faculty Development Award, and NSF grant CCR-9010730.

References

[1] H. Attiya, C. Dwork, N. Lynch and L. Stockmeyer, "Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty," *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, pp. 359–369.

[2] E. Arjomandi, M. Fischer and N. A. Lynch, "Efficiency of Synchronous versus Asynchronous Distributed Systems," *Journal of the ACM*, Vol. 30, No. 3, 1983, pp. 449–456.

[3] H. Attiya and N. A. Lynch, "Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty," *Proc. 10th Real-Time Systems Symp.*, Dec. 1989, pp. 268–284.

[4] H. Attiya and M. Mavronicolas, "Efficiency of Semi-Synchronous versus Asynchronous Networks," *Proc. 28th Allerton Conf. on Communications, Control and Computing*, Oct. 1990, pp. 578–587.

[5] B. A. Coan and G. Thomas, "Agreeing on a Leader in Real-Time," *Proc. 11th Real-Time Systems Symp.*, Dec. 1990.

[6] B. A. Coan and J. L. Welch, "Transaction Commit in a Realistic Timing Model," *Distributed Computing*, Vol. 4, 1990, pp. 87–103.

[7] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the ACM*, Vol. 34, No. 1, Jan. 1987, pp. 77–97.

[8] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *SIAM Journal of Computing*, Vol. 12, No. 13, Nov. 1983, pp. 656–666.

[9] K. Jeffay, "The Real-Time Producer/Consumer Paradigm: A Paradigm for the Efficient, Predictable Real-Time System," University of North Carolina at Chapel Hill, Department of Computer Science, submitted for publication. April 1991.

[10] K. Jeffay, D. F. Stanat and C. U. Martel, "On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. 12th IEEE Real-Time Systems Symp.*, Dec. 1991, pp. 129–139.

[11] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, Jan. 1973, pp. 46–61.

[12] M. Mavronicolas, "Efficiency of Semi-Synchronous versus Asynchronous Systems: Atomic Shared Memory," TR-03-92, Aiken Computation Lab., Harvard University, Jan. 1992.

[13] S. Ponzio, "Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures," *Proc. ACM Symp. on Principles of Distributed Computing*, Oct. 1991, pp. 125–137.

[14] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. 9th ACM Symp. on Theory of Computing*, 1977, pp. 91–97.

The Possibility and the Complexity of Achieving Fault-Tolerant Coordination*

Rida Bazzi†
Gil Neiger
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

The problem of fault-tolerant coordination is fundamental in distributed computing. In the past, researchers have considered two types of coordination: *general coordination*, in which the actions of faulty processors are irrelevant, and *consistent coordination*, in which the faulty processors are forbidden from acting inconsistently. This paper studies the possibility and complexity of achieving coordination in synchronous and asynchronous systems with crash, send-omission, and general omission failures. We indicate the systems in which coordination cannot be achieved and, when it can, analyze the computational complexity of optimally achieving it. In some cases, optimum solutions can be implemented in polynomial time, while in others they require NP-hard local computation. These results provide a thorough characterization of coordination and will thus aid researchers in determining the approach to take when attempting to achieve fault-tolerant coordination.

1 Introduction

Coordinating the activity of the processors in a distributed system is a fundamental problem in distributed computing. In such problems, processors are required to agree on a common action to perform and to ensure that the action chosen is valid given the context within which they are operating. This paper specifically considers *fault-tolerant* coordination and assumes that some (but not all) of the processors in a system may be faulty. Fault-tolerant coordination requires that the nonfaulty processors successfully coordinate their actions despite the failures of others. There is a large body of literature within computer science that has studied fault-tolerant solutions to coordination problems, such as *Reliable Broadcast* and *Distributed Consensus* (Fischer [3] provides a survey of many such problems).

This paper considers a broad range of coordination problems and divides them into two classes. The first is the class of the *general* coordination problems. These require agreement only among the correct processors. Most coordination problems considered in the literature are in this class. The second class is that of *consistent* coordination problems. When a faulty processor performs an action, these problems constrain it to perform one that is consistent with those performed by the correct processors.

This paper explores solutions to such problems within several types of distributed systems. We consider systems with both synchronous and asynchronous message passing. We also consider systems with crash (stopping), send-omission, and general (send-receive) omission failures. For each system, our goal is to determine the cases in which such problems can be solved and, in those cases, to find the best possible solutions. These are solutions that are optimum in the number of rounds of communication needed. For such solutions, we analyze the time-complexity of the *local* computation they require.

The results in this paper are based on the relationship between coordination and different forms of *processor knowledge* [8]. It is well-established

*This work was supported in part by the National Science Foundation under grants CCR-8909663 and CCR-9106627.
†This author was supported in part by a scholarship from the Hariri Foundation.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0203...$1.50

that knowledge can be used to characterize and improve solutions to various problems in distributed computing [1,7,9,10,11]. For example, Moses and Tuttle [12] showed that a weak form of *common knowledge* is necessary for the solution of general *simultaneous* coordination problems and used this fact to derive optimum solutions to such problems. Neiger and Tuttle [15] subsequently showed related results considering consistent coordination and a stronger form of common knowledge. In a previous paper [13], we considered nonsimultaneous coordination problems (both general and consistent), characterized their solutions in terms of several types of knowledge, and explored the space of optimum solutions.

In this paper, we concentrate on *eventual common knowledge* [8,16]. We do so because earlier work [13] showed that eventual common knowledge is necessary whenever coordination is achieved. Thus, if some form of eventual common knowledge is impossible in a system, the same is true of the corresponding form of coordination. If testing for eventual common knowledge is computationally expensive, the same is true of optimum coordination protocols. Reasoning about eventual common knowledge can be difficult because its semantic structure is more complex than that of simple common knowledge; Tuttle [16] showed that its semantics correspond to an optimal strategy in game theory. However, we have been able to show that, for the cases we consider, eventual common knowledge is closely related to *distributed knowledge* [2,8], which is simple knowledge ascribed to a group rather than a single processor. Reasoning about distributed knowledge is relatively easy, and it is by using distributed knowledge that we prove many of the results of this paper.

The important contributions of this paper are the following.

- For some systems in which as many as half of the processors may fail, achieving consistent coordination is impossible because the necessary form of eventual common knowledge cannot be achieved. These systems are synchronous systems with general (send-receive) omission failures and asynchronous systems with send- or general omission failures. Intuitively, the impossibility results stem from the potential confusion as to the identity of the correct processors.

- We provide a thorough analysis of the relationship between eventual common knowledge and distributed knowledge as relevant to the problem of achieving fault-tolerant coordination.

- For most systems with general omission failures in which consistent coordination is possible (i.e., in which the correct processors are always a majority), optimum solutions require NP-hard local computation. This is shown by proving the NP-hardness of testing for distributed knowledge.

Because these results provide a thorough characterization of coordination, they can greatly aid researchers in determining the approach to take when attempting to achieve fault-tolerant coordination.

2 Definitions

This section defines a model of a distributed system. This model is similar to others used to study knowledge and coordination [1,8,9,12,15].

A distributed system consists of a finite set P of n processors; they are connected by a communication network such that any processor can send a message to any other. All processors share a clock that starts at time 0 and advances in increments of one.[1] Before a computation begins, each processor may have an initial external input. Computation then proceeds in a sequence of *rounds*, with round r taking place between time $r-1$ and time r. In every round, a processor sends messages to other processors, receives messages that have arrived since the last round, performs some local computation and, optionally, a coordination action. At any given time, a processor's state consists of the time on the global clock, the messages it has sent and received, its initial input, and the coordination actions it has performed. It is assumed that all messages are tagged with the times at which they were sent and received and that coordination actions tagged with the time when they were performed. A *global state* is a tuple $\langle s_1, \ldots, s_n \rangle$ of local states.

Processors follow a *communication protocol*, which specifies the messages a processor is required to send for a round as a function of the processor's local state at the beginning of that round. A *run* is a protocol paired with an infinite sequence of global states, one per round. An ordered pair $\langle r, l \rangle$, where r is a run and l is a natural number, is called a *point*

[1] Note that we are using a round based model of asynchronous systems; that is, we assume that processors have access to a global clock. This is done purely for ease of presentation. The full paper defines a more general mode for asynchronous systems.

and represents the system after the first l rounds of r. The local state of processor p at that point is denoted by $r_p(l)$. Some processors correctly follow the protocol and are thus *nonfaulty*. Let $\mathcal{N}(r)$ represent the set of processors nonfaulty in run r. Other processors may be subject to failures. We consider three types of failures. These are *crash failures* (in which a faulty processor simply stops, possibly in the middle of a round), *send-omission failures* (in which a faulty processor may intermittently omit to send messages), and *general omission failures* (in which a faulty processor may intermittently omit to send and/or receive messages). We assume that the faulty behavior in a run is completely independent of the inputs to the processors.

This paper considers *synchronous* systems, in which processors messages are always delivered in the round in which they are sent, and *asynchronous* systems, in which message-passing is reliable, but there is no bound on message delivery times. Within the scope of asynchronous systems, we consider those in which messages between two processors are always delivered in the order sent (FIFO communication) as well as those in which this is not true.

This work identifies a system with the set of all runs of a communication protocol under a specified set of assumptions about synchrony and failures. In order to analyze systems, it is convenient to have a logical language to make statements about the system. A *fact* in this language is interpreted to be a property of points: a fact φ will be either true or false at a given point $\langle r,l \rangle$, denoted $(r,l) \models \varphi$ and $(r,l) \not\models \varphi$, respectively. Fact φ is *valid in a system* if it is true at all points in the system. Although facts are interpreted as properties of points, it is often convenient to refer to facts that are about objects other than points (e.g., properties of runs). In general, a fact φ is a *fact about* X if fixing X determines the truth (or falsity) of φ.

3 Coordination Problems

This section defines two classes of coordination problems. In general, a *coordination problem* is a finite set of actions $C = \{a_1, \ldots, a_m\}$. Each action has associated with it an *enabling condition* ok_i, which is a fact about the input and the identities of the faulty processors (thus, it is a fact about runs). The processors must coordinate to choose a common action that is enabled. Processors *need not* perform their actions simultaneously.

To solve a coordination problem, we augment a communication protocol with a set of functions that tell a process when to perform a coordination action; we call the result an *action protocol*. Actions protocols must be such that no processor performs more than one action. Informally, an action protocol is correct if, in all runs, processors must agree and must choose an action that is enabled. We consider two classes of such protocols. In one class, the actions taken by the *faulty* processors are not relevant; in the other, their actions are subject to the same correctness criteria as those of the nonfaulty processors. We call these classes *general* and *consistent*, respectively. Formally, *a protocol generally satisfies C* (or *G-satisfies C*) if the following conditions hold of all runs of the protocol:

1. Validity. If an action is performed by a nonfaulty processor, then that action is enabled.

2. Agreement. If a nonfaulty processor performs an action, then all nonfaulty processors perform that action.

A protocol consistently satisfies C (or *C-satisfies C*) if the above conditions hold with the first occurrence of the word "nonfaulty" omitted from each. Earlier literature on coordination concentrated on general coordination [9,12]; more recently, researchers have begun to consider consistent coordination [13,15]. Note that these coordination problems *can* be solved in asynchronous systems in spite of the impossibility results of Fischer et al. [4]. This is because they do not require nonfaulty processors to perform an action in every run.

Solutions to coordination problems are compared by comparing their behavior in corresponding runs. Two runs correspond if they have the same initial input and the same pattern of faulty behavior. Protocol P_1 *dominates* P_2 if, in any pair of corresponding runs, no processor performs an action later in the run of P_1 than it does in the run of P_2. A protocol is *X-optimum for C* (where X is either G or C) if it X-satisfies C and dominates every other protocol that does so. Some problems do not have optimum solutions. For example, Moses and Tuttle [12] showed that *Eventual Byzantine Agreement* does not have an optimum solution. Nevertheless, there are coordination problems for which optimum solutions do exist. Neiger and Bazzi [13] gave a precise characterization of these problems. They also considered *optimal* solutions to coordination problems. Protocol P is *X-optimal for C* if it X-satisfies C and if every P' that X-satisfies C and dominates P is in turn dominated by P.

4 Definitions of Knowledge

Processor knowledge was first defined by Halpern and Moses [8] in the following way. *Processor p knows φ at point (r,l)*, denoted $(r,l) \models \mathsf{K}_p\varphi$, if $(r',l) \models \varphi$ for all runs r' such that $r'_p(l) = r_p(l)$ (recall that the global clock is always part of a processor's local state). It is often useful to condition a processor's knowledge on its being nonfaulty. We say that *processor p believes φ* if p knows that, if it is in $\mathcal{N}$, φ is true. That is, $\mathsf{B}_p\varphi \equiv \mathsf{K}_p(p \in \mathcal{N} \Rightarrow \varphi)$. It is easy to see that $(r,l) \models \mathsf{B}_p\varphi$ if $(r',l') \models \varphi$ for all runs r' such that $r'_p(l) = r_p(l)$ and $p \in \mathcal{N}(r')$. Processor knowledge, using the K_p operators, will be used to define strong notions of group knowledge, while processor belief, using B_p, will be used to define weaker notions.

Because this paper deals with coordination among the nonfaulty processors, we are specifically interested in the knowledge possessed by the set $\mathcal{N}$ of nonfaulty processors. *Everyone in $\mathcal{N}$ knows φ*, denoted $\mathsf{E}_\mathcal{N}\varphi$, is defined as $\bigwedge_{p\in\mathcal{N}} \mathsf{K}_p\varphi$. *All processors in $\mathcal{N}$ believe φ*, denoted $\mathsf{A}_\mathcal{N}\varphi$, is $\bigwedge_{p\in\mathcal{N}} \mathsf{B}_p\varphi$. *Strong common knowledge* ($\mathsf{S}_\mathcal{N}\varphi$) is equivalent to the infinite conjunction $\bigwedge_{i\geq 1} \mathsf{E}_\mathcal{N}{}^i\varphi$, while *weak common knowledge* ($\mathsf{W}_\mathcal{N}\varphi$) is equivalent to $\bigwedge_{i\geq 1} \mathsf{A}_\mathcal{N}{}^i\varphi$.

Eventual common knowledge [8,16] relaxes the simultaneity inherent in common knowledge. For this reason, it is more appropriate in the study of problems that do not require simultaneous coordination. Informally, a fact is eventual common knowledge to a set of processors if they all eventually know it, all eventually know that all others eventually know it, and so on *ad infinitum*. As we showed elsewhere [13], eventual common knowledge is necessary for achieving termination in solving a coordination problem. The definition of eventual knowledge uses the temporal operator *eventually* $\Diamond$. We say $(r,l) \models \Diamond\varphi$ if and only if $(r,l') \models \varphi$ for some $l' \geq l$. Eventual common knowledge is also defined using fixed points. *Strong eventual common knowledge of fact φ by set $\mathcal{N}$*, denoted $\mathsf{S}^\Diamond_\mathcal{N}\varphi$, is the greatest fixed point of the equivalence $X \Leftrightarrow \Diamond\mathsf{E}_\mathcal{N}(\varphi \wedge X)$.[2] *Weak eventual common knowledge of fact φ by set $\mathcal{N}$*, denoted $\mathsf{W}^\Diamond_\mathcal{N}\varphi$, is the greatest fixed point of the equivalence $X \Leftrightarrow \Diamond\mathsf{A}_\mathcal{N}(\varphi \wedge X)$. In general, $\mathsf{S}^\Diamond_\mathcal{N}\varphi$ implies (but is not necessarily equivalent to) the infinite conjunction $\bigwedge_{i\geq 1}(\Diamond\mathsf{E}_\mathcal{N})^i\varphi$ and a similar statement is true for weak eventual common knowledge. One should note that eventual common knowledge is *weaker* that simple common knowledge. It does not require that processors gain their knowledge simultaneously or that all levels of knowledge will ever hold simultaneously. Eventual common knowledge does not, in general, imply "eventually" common knowledge.

If a fact is eventual common knowledge to a set, then all members of the set eventually know (or believe) this. That is, both $\mathsf{S}^\Diamond_\mathcal{N}\varphi \Rightarrow \Diamond\mathsf{E}_\mathcal{N}\mathsf{S}^\Diamond_\mathcal{N}\varphi$ and $\mathsf{W}^\Diamond_\mathcal{N}\varphi \Rightarrow \Diamond\mathsf{A}_\mathcal{N}\mathsf{W}^\Diamond_\mathcal{N}\varphi$ are valid. Each form of eventual common knowledge satisfies an induction rule that can be used to show that certain facts are eventual common knowledge:

- If $\varphi \Rightarrow \Diamond\mathsf{E}_\mathcal{N}(\varphi \wedge \psi)$ is valid in a system, then $\varphi \Rightarrow \mathsf{S}^\Diamond_\mathcal{N}\psi$ is also valid in that system.
- If $\varphi \Rightarrow \Diamond\mathsf{A}_\mathcal{N}(\varphi \wedge \psi)$ is valid in a system, then $\varphi \Rightarrow \mathsf{W}^\Diamond_\mathcal{N}\psi$ is also valid in that system.

Because the set $\mathcal{N}$ is assumed to be nonempty, it is not hard to see that $\mathsf{S}^\Diamond_\mathcal{N}\varphi \Rightarrow \varphi$ and $\mathsf{W}^\Diamond_\mathcal{N}\varphi \Rightarrow \varphi$ are valid when φ is fact about runs.

We now present two theorems regarding the relationship between coordination and eventual common knowledge that we proved in an earlier paper [13]. The first theorem shows that some form of eventual common knowledge is necessary to achieve coordination.

The second theorem shows that any *optimum* coordination problem *must* perform an action as soon as some enabling condition becomes eventual common knowledge:

Theorem 1: *Let C be a coordination problem. If a protocol G-satisfies C, then, whenever processor p performs action a_i, $\mathsf{B}_p\mathsf{W}^\Diamond_\mathcal{N} ok_i$. That is, p must know that, if it is nonfaulty, ok_i is weak eventual common knowledge. Similarly, if a protocol C-satisfies C, then, whenever processor p performs action a_i, $\mathsf{K}_p\mathsf{S}^\Diamond_\mathcal{N} ok_i$.*

Theorem 2: *Let C be a coordination problem. If a protocol is G-optimum for C, then processor p must perform some action as soon as $\mathsf{B}_p\mathsf{W}^\Diamond_\mathcal{N} ok_i$ holds for any action a_i. Similarly, if a protocol is C-optimum for C, then processor p must perform some action as soon as $\mathsf{K}_p\mathsf{S}^\Diamond_\mathcal{N} ok_i$ holds for any action a_i.*

Together, these results show that a thorough understanding of eventual common knowledge will give a better understanding of the possibility of achieving coordination and of the complexity of

[2] This is slightly stronger than the original definition of Halpern and Moses [8]. They defined eventual common knowledge to be the greatest fixed point of the equivalence $X \Leftrightarrow \bigwedge_{p\in\mathcal{N}} \Diamond\mathsf{K}_p(\varphi \wedge X)$. For all cases considered in this paper, this definition is equivalent to that given here for strong eventual common knowledge.

optimum protocols. Section 5 gives situations in which consistent coordination is impossible. Section 6 considers the complexity of optimum coordination protocols.

5 Impossibility Results for Consistent Coordination

This section shows that, if $n \leq 2t$, then consistent coordination can be achieved neither in synchronous systems with general omission failures nor in asynchronous systems with send- or general omission failures. This is done by showing that it is impossible to achieve strong eventual common knowledge in those systems. This distinction between general and consistent coordination occurs because of the uncertainty regarding the identity of the correct processors. There can be two disjoint sets of at most t processors such that processors within a set communicate with no trouble, but processors in different sets never communicate. Since this behavior is consistent with either set of processors being the set of faulty processors, no processor can know whether or not it is faulty. In the context of general coordination, this is not a problem, since the actions of the faulty processors are unimportant, so each set can simply behave as if it is the set of nonfaulty processors. In the context of *consistent* coordination, however, this behavior is critical: because the processors may be isolated from important information and not know whether or not they are faulty, the correct processors can become "paralyzed" and unable to act.

This technique was first used by Neiger and Toueg [14] to show the impossibility of certain translations between systems with failures. Neiger and Tuttle [15] later used it to show the impossibility of *simultaneous* coordination in systems with general omission failures and $n \leq 2t$.

We handle the situations of synchronous and asynchronous systems differently. This is because the results for asynchronous systems apply to send-omission failures as well as general omission failures. Intuitively, such a system may be "partitioned" by a combination of "delayed" messages and send-omission failures.

5.1 Synchronous Systems with General Omission Failures

This section considers a synchronous systems with general omission failures and $n \leq 2t$. We say that two sets A and B *partition* the set of processors if A and B are nonempty and disjoint, $A \cup B = P$, and $|A|, |B| \leq t$. Given a run r such that there is complete (failure-free) communication within A and within B, let r^k be the run identical to r (the failure pattern is the same) except that, for every round $l > k$, no processor in B sends or receives a message to or from any processor in A in round l of r^k; note that A is the set of nonfaulty processors in r^k. We say that r^k is *the result of partitioning r into A and B from time k*. Informally, the following lemma states that, if a fact becomes strong eventual common knowledge in r, then it also becomes eventual common knowledge in r^k for all k.

Lemma 3: *Let φ be a fact about runs and let A and B be two sets that partition the set of processors. If $(r,l) \models \bigvee_{p \in A} \mathsf{K}_p \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$, then, for all k, there is a $j \geq k$ such that, if r^k is the result of partitioning r into A and B from time k, $(r^k, j) \models \bigvee_{p \in A} \mathsf{K}_p \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$.*

Proof: The proof is by reverse induction on k. For $k \geq l$, the result easily holds for any $j \geq k$. This is because r^k and r cannot be distinguished at time l, so $(r^k, l) \models \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$. Since φ is a fact about runs, $\mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$ is stable and knowledge of it cannot be lost. Now assume that the result holds for $k+1$. That is, $(r^{k+1}, j) \models \bigvee_{p \in A} \mathsf{K}_p \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$ for some $j \geq k+1$. Let r' be a run identical to r^{k+1} except that $\mathcal{N}(r') = B$ and all processors in A fail to send to those in B in round k. It should be clear that r' is a run of the system: there are no failures among A or among B in r^{k+1} and both have size at most t. Because the system admits general omission failures, simply change send omissions to receive omissions and vice versa. Since $r'_p(j) = r^{k+1}_p(j)$ for any $p \in \mathcal{N}(r^{k+1})$, $(r', j) \models \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$. Since eventual common knowledge eventually becomes known to every nonfaulty processor, there is some $j' \geq j$ such that $(r', j') \models \bigvee_{p \in B} \mathsf{K}_p \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$. Now let r^k be a run identical to r' except that $\mathcal{N}(r^k) = A$ and all processors in B fail to send to those in A in round k. Note that r^k is the result of partitioning r into A and B from time k. Furthermore, $(r^k, j') \models \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$, using the argument given above. This means that there is some $j'' \geq j'$ such that $(r^k, j'') \models \bigvee_{p \in A} \mathsf{K}_p \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$, completing the proof. □

Given that we can relate any run with strong eventual common knowledge to one in which the system is partitioned from time 0 (and in which the same knowledge holds), we can show the following:

Theorem 4: *If φ is a fact about the input and faulty processors that is not valid, then, for every failure-free run r and time l, $(r,l) \models \neg \mathsf{S}^{\Diamond}_{\mathcal{N}} \varphi$.*

Proof: The proof is by repeated application of Lemma 3, where the input (and set of faulty processors) is manipulated in the partitioned runs to relate the original run to one in which φ (and thus $\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$) is not true. The contradicts the fact that $\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ was true in the original run. Note that failure-free runs are candidates for the application of Lemma 3. □

A coordination problem is *nontrivial* if none of its enabling conditions is valid. It is easy to use Theorem 4 to show the following:

Theorem 5: *In synchronous systems with general omission failures and $n \leq 2t$, if a protocol C-solves a nontrivial coordination problem, then no action is performed in failure-free runs.*

5.2 Asynchronous Systems with Send- or General Omission Failures

This section shows a result analogous to that shown in the previous section. This result holds for asynchronous systems and includes send- as well as general omission failures as long as $n \leq 2t$. Because of this, we need to reconsider the definition of a partitioning, which depended on the possibility of general omission failures. Now, r^k is *a result of partitioning r into A and B from time k* if r^k is identical to r except that, for every round $l > k$, no processor in B sends to any processor in A in round l of r^k. Note that, because messages need not be delivered in the round in which they are sent, there may be partitions of a run r that differ with respect to when messages are delivered.

The following analogue to Lemma 3 holds in this case:

Lemma 6: *Let φ be a fact about runs and let A and B be two sets that partition the set of processors. If $(r,l) \models \bigvee_{p\in A} \mathsf{K}_p\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$, then, for all k, there is a $j \geq k$ and an r^k that is the result of partitioning r into A and B from time k such that $(r^k,j) \models \bigvee_{p\in A} \mathsf{K}_p\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$.*

Proof: The proof is identical to that of Lemma 3 with the following exception. When run r' is constructed, we cannot simply switch send and receive omissions, as there are only send omissions. Instead, let r' be such that messages omitted from B to A arrive after time j. Again, $r'_p(j) = r_p^{k+1}(j)$ for any $p \in \mathcal{N}(r^{k+1})$. The same method can be used when constructing r^k. □

Lemma 6 can now be used to prove an analogue to Theorem 4, giving in the end the following result:

Theorem 7: *In asynchronous systems with send- or general omission failures and $n \leq 2t$, if a protocol solves a nontrivial coordination problem, then no action is performed in failure-free runs.*

6 Complexity Results for Optimum Coordination

This section consider systems in which coordination problems can be solved. Thus, it considers general coordination as well as consistent coordination in systems other than those discussed in Section 5. It begins by relating the two forms of eventual common knowledge to another form of knowledge that is easier to reason about: *distributed knowledge*. It then considers the complexity of testing for distributed knowledge. In some cases, we can show that the necessary tests can be performed in polynomial time. In others, we show that these tests are NP-hard. Because of the proven relationship between distributed and eventual common knowledge, these results also hold for the implementation of optimum coordination protocols.

The results given in this section apply primarily to coordination problems whose enabling conditions are facts about the input. Note that, if φ is a fact about the input, then $\mathsf{B}_p\varphi \Leftrightarrow \mathsf{K}_p\varphi$. To study the complexity of coordination, one must define the parameters that determine the *size* of a problem. We consider this to be the number of processors in the system and the number of rounds that have taken place.

The results below apply to full information protocols. These are protocols in which processors communicate all the information available to them every round. Moses and Tuttle [12] showed that full information protocols can be implemented with polynomial size messages in synchronous systems with the types of failure we are considering. This generalizes to asynchronous systems as follows. After round r, a labeled communication graph is a graph with $(r+1)n$ vertices arranged in $r+1$ columns of n vertices each. Each row corresponds to one of the processors. An edge between vertices (p_1,r_1) and (p_2,r_2) means that the message sent by p at the beginning of round r_1 is received by q at the end of round $r_2 - 1$. Processors maintain a local communication graph that includes its view of the execution and sends this graph to all processors in each round. When one processor receives the communication graph from another, it updates

its communication graph by forming the union of the two graphs and adding an edge corresponding to the message received.

In the case of general coordination (Section 6.2), we make the standard assumption that the truth or falsity of a fact can be calculated in polynomial time from the labeled communication graph.

6.1 Eventual Common Knowledge and Distributed Knowledge

We begin by giving a definition of distributed knowledge [2,8]. This is simply a group knowledge analogue to the single processor knowledge. Fact φ is *distributed knowledge* to the nonfaulty processors at point $\langle r,l\rangle$, denoted $(r,l) \models \mathsf{D}_{\mathcal{N}}\varphi$, if $(r',l) \models \varphi$ for all runs r' such that $r'_p(l) = r_p(l)$ for all $p \in \mathcal{N}(r)$. Thus, the combined states of the processors nonfaulty at $\langle r,l\rangle$ determine that φ holds. Because it may be possible that no processor in $\mathcal{N}(r)$ explicitly knows φ, we say that the knowledge is distributed among the members of the group. Note first the following fact relating knowledge, belief, and distributed knowledge:

Lemma 8: *If φ is a fact about runs, then $\mathsf{K}_p\varphi \Rightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$ is valid.*

It turns out that distributed knowledge is closely related to eventual common knowledge. We begin by noting that it implies weak eventual common knowledge because we can prove the following:

Lemma 9: *For any fact φ about runs, $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi$ is valid in any system running a full-information protocol.*

Proof: We will prove this using the induction rule for weak eventual common knowledge. Specifically, we will prove that $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{A}_{\mathcal{N}}(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$ is valid in the system; by induction, the desired implication will then be valid. Assume that $(r,l) \models \mathsf{D}_{\mathcal{N}}\varphi$. It suffices to show that, for any processor $p \in \mathcal{N}$, there is some time $l' \geq l$ such that $(r,l') \models \mathsf{B}_p(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$. Let l' be the earliest time by which p has received a message from each processor in $\mathcal{N}$ that was sent at time l or later. Since message passing is reliable between correct processors, l' is guaranteed to exist. Because $(r,l) \models \mathsf{D}_{\mathcal{N}}\varphi$ and φ is a fact about runs, $(r,l') \models \mathsf{K}_p\varphi$; by time l', p has all the information available to the nonfaulty processors at time l. It should be clear that $\mathsf{K}_p\varphi \Rightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$ in any system, so $(r',l) \models \mathsf{B}_p(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$, as desired. □

We next prove that belief or knowledge of eventual common knowledge implies belief or knowledge of distributed knowledge.

Lemma 10: *Let φ be a fact about the input. Then the following are valid in any system: $\mathsf{B}_p\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi \Rightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$ and $\mathsf{K}_p\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi \Rightarrow \mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$.*

Proof: The proof is given only for the first case. Let $(r,l) \models \mathsf{B}_p\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi$. Since $\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi \Rightarrow \varphi$ is valid for facts about runs, $(r,l) \models \mathsf{B}_p\varphi$. Since φ is a fact about the input, $\mathsf{B}_p\varphi \Leftrightarrow \mathsf{K}_p\varphi$ is valid, so $(r,l) \models \mathsf{K}_p\varphi$. By Lemma 8, $(r,l) \models \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$, as desired. □

The converse of the first implication follows from Lemma 9. The converse of the second is valid for all systems that we consider. Specifically, it holds in all cases except for systems in which consistent coordination is impossible and we can thus prove Lemmas 11 and 12:

Lemma 11: *Consider the execution of a full information protocol in a synchronous system with crash or send-omission failures or with general omission failures and $n > 2t$. If φ is a fact about runs, then $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid.*

Proof: By the induction rule for strong common knowledge, it suffices to show that $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{E}_{\mathcal{N}}(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$. Assume that $(r,l) \models \mathsf{D}_{\mathcal{N}}\varphi$. Consider now two cases:

- There are only crash or send-omission failures. Because all nonfaulty processors send correctly in round $l+1$, all noncrashed processors will know φ by the end of that round. In round $l+2$, every nonfaulty processor will, for every other processor, receive a message from that processor or know that it is faulty. That is, $(r,l+2) \models \mathsf{E}_{\mathcal{N}} \bigwedge_{p \in P}(p \notin \mathcal{N} \vee \mathsf{K}_p\varphi)$, which means $(r,l+2) \models \mathsf{E}_{\mathcal{N}}(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$, giving the desired result.
- There are general omission failures and $n > 2t$. Because all nonfaulty processors communicate correctly in round $l+1$, $(r,l+1) \models \mathsf{E}_{\mathcal{N}}\varphi$. In round $l+2$, each nonfaulty processor receives messages from at least $n - t > t$ processors that know φ. Since it knows that at least one of these must be correct, $(r,l+2) \models \mathsf{E}_{\mathcal{N}}(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$, as desired.

In either case, $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{E}_{\mathcal{N}}(\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$ so, by induction, $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid. □

Lemma 12: *Consider the execution of a full information protocol in an asynchronous system with crash failures or with send- or general omission failures and $n > 2t$. If φ is a fact about runs, then $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid.*

Proof: The proof begins by showing that $\Diamond\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid. We will do this by induction, showing first that $\Diamond\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{E}_{\mathcal{N}}(\Diamond\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$. Let $(r,l) \models \Diamond\mathsf{D}_{\mathcal{N}}\varphi$. Let $k \geq l$ be such that $(r,k) \models \mathsf{D}_{\mathcal{N}}\varphi$ and let $j > k$ be such that every nonfaulty processor receives a message from every other that was sent after time k. Clearly, $(r,j) \models \mathsf{E}_{\mathcal{N}}\varphi$. Now consider two cases

- There are only crash failures. Consider any $p \in \mathcal{N}(r)$. At time $j+1$, p knows that it correctly sent messages to all nonfaulty processors after it knew φ. Since these must be eventually received, $(r,j+1) \models \mathsf{K}_p\Diamond\mathsf{D}_{\mathcal{N}}\varphi$. This means that $(r,l) \models \Diamond\mathsf{E}_{\mathcal{N}}(\Diamond\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$.
- There are send- or general omission failures and $n > 2t$. Let $j' > j$ be such that every nonfaulty processor receives a message from every other that was sent after time j. Since $n > 2t$, each nonfaulty processor receives by time j at least $n - t > t$ such messages and knows that at least one is from a nonfaulty processor. Thus, $(r,j') \models \mathsf{E}_{\mathcal{N}} \bigvee_{p\in\mathcal{N}} \mathsf{K}_p\varphi$, which means that $(r,j') \models \mathsf{E}_{\mathcal{N}}\Diamond\mathsf{D}_{\mathcal{N}}\varphi$. Thus, $(r,l) \models \Diamond\mathsf{E}_{\mathcal{N}}(\Diamond\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$, as desired.

In either case, $\Diamond\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{E}_{\mathcal{N}}(\Diamond\mathsf{D}_{\mathcal{N}}\varphi \wedge \varphi)$ is valid so, by induction, $\Diamond\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid. Since $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \Diamond\mathsf{D}_{\mathcal{N}}\varphi$ is valid, $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid as well. □

Lemmas 11 and 12 lead to the equivalences noted in the following theorem:

Theorem 13: *Let φ be a fact about the input. Then $\mathsf{B}_p\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi \Leftrightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$ is valid. In any system in which consistent coordination is possible, $\mathsf{K}_p\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi \Leftrightarrow \mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ is valid*

Proof: A proof is given only for the second conclusion. Since $\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi \Rightarrow \mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi$ is valid, Lemma 10 implies that $\mathsf{K}_p\mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi \Rightarrow \mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ is also valid. By Lemmas 11 and 12, $\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{S}^{\Diamond}_{\mathcal{N}}\varphi$ is valid for all systems in which consistent coordination is possible. This completes the proof. □

These equivalences are important because they show that the belief or knowledge needed for coordination can be exactly expressed in terms of distributed knowledge. Because distributed knowledge is easy to reason about, these equivalences allow us to analyze the complexity of optimum coordination protocols.

6.2 The Complexity of Optimum General Coordination

This section and Section 6.3 both restrict their consider to coordination problems for which the enabling conditions are facts about the input. Theorem 13 showed that, for such facts, $\mathsf{B}_p\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi \Leftrightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$. Theorem 2 showed that, in any optimum protocol for general coordination, processor p performs an action as soon as $\mathsf{B}_p\mathsf{W}^{\Diamond}_{\mathcal{N}}\varphi$. Thus, we can address the complexity of optimum general coordination by seeing how hard it is for p to determine $\mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$. It turns out that this can be done relatively easily.

Because φ is a fact about the input, $\mathsf{B}_p\varphi \Rightarrow \mathsf{K}_p\varphi$. As noted earlier, $\mathsf{K}_p\varphi \Rightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$; thus, $\mathsf{B}_p\varphi \Rightarrow \mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi$. It is obvious that $\mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi \Rightarrow \mathsf{B}_p\varphi$. Thus, $\mathsf{B}_p\mathsf{D}_{\mathcal{N}}\varphi \Leftrightarrow \mathsf{B}_p\varphi$, meaning that a processor must act as soon as it believes φ. This is very easy for facts about the input. A processor simply examines its local state and determines whether or not the inputs of which it is aware support φ. This can be done in polynomial time.

6.3 The Complexity of Optimum Consistent Coordination

Understanding the complexity of consistent coordination is not as easy doing so for general coordination. In these cases, processors need to test for *knowledge* of, and not *belief* in, distributed knowledge of the enabling conditions. It turns out that, in some cases, this can be checked in polynomial time, while in others checking is NP-hard. We begin by considering cases in which doing so is easy.

6.3.1 When Optimum Consistent Coordination is Easy

This section shows that, in systems in which failures are easily detected or in which they cannot be detected at all, testing for knowledge of distributed knowledge can be done efficiently. As will be seen in the next section, these tests are NP-hard in systems with general omission failures, in which the identity of faulty processors is hard to verify. The remainder of this section considers coordination problems in which the enabling conditions are facts about the input that can be expressed in a certain form. We say that a fact about the input is *basic* if it is of the

form "p's initial input is (not) v." A formula φ is in conjunctive normal form (CNF) if it is of the form $\bigwedge_i \bigvee_j \psi_{i,j}$ (where i and j have finite range). While any fact about the input can expressed in CNF, we restrict our consideration to those that can be so expressed in length polynomial in the number of processors in the system.[3]

As an example, consider a synchronous system with send-omission failures. Processor p knows processor q is faulty if and only if p's local state includes indication of a missing message that q should have sent. This can be checked easily by inspecting p's communication graph. Suppose that p knows of $f \leq t$ such failures. Recall that φ is in CNF; that is, $\varphi = \bigwedge_i \bigvee_j \psi_{i,j}$, where each $\psi_{i,j}$ is a basic fact about the input. Whether or not a processor knows a basic fact can be easily checked by inspecting the communication graph. It is not hard to see that $\mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ if and only if $\bigwedge_i \mathsf{K}_p\mathsf{D}_{\mathcal{N}} \bigvee_j \psi_{i,j}$; thus, we can test for each $\mathsf{K}_p\mathsf{D}_{\mathcal{N}} \bigvee_j \psi_{i,j}$ separately. For each i, test as follows. Check to see if $\bigvee_j \psi_{i,j}$ is valid; this requires only checking to see if the conjunction contains a basic fact and its negation. If the disjunction is valid, then $\mathsf{K}_p\mathsf{D}_{\mathcal{N}} \bigvee_j \psi_{i,j}$ trivially holds. If not, inspect the communication graph and count the number of processors (that are not known to be faulty) that know one of the basic facts that comprise the disjunction. If this number is greater than $t - f$, then at least one of these is correct and $\mathsf{K}_p\mathsf{D}_{\mathcal{N}} \bigvee_j \psi_{i,j}$ holds; otherwise, all these processors could be faulty and $\neg\mathsf{K}_p\mathsf{D}_{\mathcal{N}} \bigvee_j \psi_{i,j}$.

In asynchronous systems without FIFO communication, it is impossible to ever detect that a processor is faulty. Any omission failure, either to send or to receive, cannot be distinguished from a message that is in transit. If communication is FIFO, then some send-omission failures can be detected (if a message arrives that is subsequent to one that was omitted). In this case, checking for knowledge of distributed knowledge is again easy using the method outline above. General omission failures cannot easily be detected (in either synchronous or asynchronous systems), and the problem becomes NP-hard; see the next section.

These observations lead to the following results:

Lemma 14: *Consider any system with crash or send-omission failures or any asynchronous system with non-FIFO communication. Let φ be a fact about the input that can be expressed in CNF in polynomial size. Then $\mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ can be checked in polynomial time.*

Theorem 15: *Consider any system with crash or send-omission failures or any asynchronous system with non-FIFO communication. Let C be a coordination problem that has a C-optimum solution and whose enabling conditions are facts about the input that can be expressed in CNF form in polynomial size. Then the C-optimum protocol can be implemented in polynomial time.*

6.3.2 When Consistent Coordination is Hard

This section considers the case of synchronous systems with general omission failures and $n > 2t$. In this case, testing for $\mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ is NP-hard even when φ is a basic fact about the input. This can be used to show that implementing optimum coordination protocols requires NP-hard computation.

This result is similar to one of Moses and Tuttle [12], who showed that optimum protocols for general simultaneous coordination required NP-hard computation; Neiger and Tuttle [15] extended this result to consistent simultaneous coordination. However, note that, in the cases considered here (nonsimultaneous coordination), the NP-hardness results apply only to consistent coordination. As noted in Section 6.2, optimum protocols for many general coordination problems can be implemented in polynomial time.

The complexity result is achieved by giving a polynomial-time Turing reduction from *clique* to testing for $\mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$. The clique problem is the following: given a graph $G = (V, E)$ and constant k $(1 \leq k \leq |V|)$, is there a clique in G of size k? This problem is NP-complete [5]. The idea behind the proof is that the communication structure of the set of nonfaulty processors in a system forms a clique. The following states the result for synchronous systems.

Lemma 16: *Consider a synchronous systems with general omission failures. Given a processor p, its local state, and some fact φ about the input and faulty processors, determining whether or not $\mathsf{K}_p\mathsf{D}_{\mathcal{N}}\varphi$ is NP-hard.*

Proof: Here is the high-level reduction:

```
for i = 2 to k do
    convert G to a system execution
    if K_q D_N φ            /* no clique of size i */
        return(NO)
return(YES)
```

[3] The restriction on polynomial length makes sense only under the assumption that any coordination problem is parameterized by n and t.

The following describes the details of the conversion and the construction of the fact φ and then proves that the overall reduction is correct.

Consider the conversion at iteration i. At this point, it is certain that G contains a clique of size $i-1$. Consider a system with $n = |V| + 2$ processors with as many as $t = n - i$ general omission failures. Thus, there must be at least $n - t = i$ nonfaulty processors. For each $v \in V$, there is a corresponding processor p_v. In addition, there are two processors q and r. Let φ be "r's initial input is 0." The following execution of a full-information protocol then occurs (only significant communication is mentioned). Processor r's initial input is 0. In round 1, q sends to all processors and r sends to none. All processors p_v ($v \in V$) send to q and r and receive from q. If $(v,w) \in E$, then p_v and p_w communicate correctly in round 1. If $(v,w) \notin E$, then no communication passes between p_v and p_w in round 1. In round 2, q receives messages from all processors. We claim that $\mathsf{K}_q\mathsf{D}_{\mathcal{N}}\varphi$ at time 2 if and only if there is no clique in G of size i.

Suppose that there is a clique in G of size $i = n - t$. Then, as far as q can tell at time 2, the corresponding processors might be the only nonfaulty ones in the system, meaning that q and r are both faulty. Since q and r are the only processors that q knows to have any knowledge about r's input, it cannot be that q knows that φ is distributed knowledge to the nonfaulty processors. Now assume that there is no clique in G of size i (recall that there must be a clique of size $i-1$). Then, at time 2, q knows that there are at most $i-1$ correct processors corresponding to vertices in the graph. It knows that r is faulty (as it sent to no processors in round 1), so q knows itself to be correct. Thus, since q knows φ, it knows that φ is distributed knowledge to the nonfaulty processors. □

Similar reasoning can be used in asynchronous systems with FIFO communication. The constructed execution is longer because, in such systems, a message subsequent to an omitted message must arrive before any failure can be detected.

The NP-hardness of testing for distributed knowledge imply an NP-hardness for consistent coordination:

Theorem 17: *Consider a system with general omission failures that is either synchronous or asynchronous with FIFO communication. Processors perform NP-hard local computation in any optimum protocol for a consistent coordination problem.*

7 Discussion and Conclusions

This paper considered the problem of fault-tolerant coordination in distributed systems. In some cases, it was determined that such coordination was impossible. In others, the computational complexity of optimum coordination protocols was analyzed. The main results are given in Table 1. Note that we provide no impossibility results for general coordination; furthermore, optimum protocols when they exist, are tractable. For consistent coordination, in which faulty processors are forbidden from taking inconsistent actions, the situation is much more complex. This is especially true for general omission failures. Here, consistent coordination is impossible if as many as half the processors may fail ($n \leq 2t$); if a majority must remain correct ($n > 2t$), then the problems may be solved, but any optimum protocol requires NP-hard local computation. For asynchronous systems, the impossibility result extends to systems with send-omission failures. The complexity results, however, do not hold unless message passing is FIFO.

Some of these results are related to earlier results shown for *simultaneous* coordination [12,15]. However, there are some important differences. One of these is the case of general coordination in synchronous systems with general omission failures and $n > 2t$. For simultaneous coordination, optimum solutions in these systems required NP-hard local computation regardless of whether coordination was general or consistent. In this paper, however, we see that there are polynomial time solutions for the nonsimultaneous general coordination but not for consistent coordination. This is the first case in the literature in which the computational complexity of optimum solutions depends on the general/consistent distinction.

Our results for coordination in asynchronous systems apply to certain algorithms of Gopal and Toueg [6]. For example, they present a solution to a problem they call *Single Value Agreement* that tolerates general omission failures when $n > 2t$. If communication is not FIFO, then their solution is optimal. If communication is FIFO, it is not; informally, this is because the knowledge that communication is ordered can allow processors to decide earlier. However, doing so optimally requires processors to perform NP-hard local computation. Gopal and Toueg did not consider FIFO communication and presented only protocols that used polynomial computation.

The complexity results of Section 6 apply to optimum solutions to coordination problems whose

Table 1: The Possibility and Complexity of Coordination

	General	Consistent	
Synchronous	polynomial time	polynomial time	Crash, Send-Omission
		NP-hard	General Omission, $n > 2t$
		impossible	General Omission, $n \leq 2t$
Asynchronous	polynomial time	polynomial time	Crash
		polynomial time	Either omission, $n > 2t$, non-FIFO
		polynomial time	Send-omission $n > 2t$, FIFO
		NP-hard	General Omission $n > 2t$, FIFO
		impossible	Either omission, $n \leq 2t$

enabling conditions are facts about the input. This was done by relating knowledge of eventual common knowledge to knowledge about distributed knowledge. For asynchronous systems (without FIFO communication), these results can be extended to include facts about failures as well. We are currently exploring ways in which such extensions can be made for other systems. We have already established a different relation between knowledge of eventual common knowledge and of distributed knowledge; we plan to extend the results of this paper to coordination problems that also depend on facts about failures.

Finally, recall that Moses and Tuttle [12] showed that some coordination problems have no optimum solution. In an earlier paper [13], we used a new form of knowledge, called *extended common knowledge*, to construct *optimal* solutions to any coordination problem. In the future, we plan to explore the complexity of this type of knowledge to better understand the complexity of these optimal solutions and thus complement the results presented here for optimum solutions.

Acknowledgements

We would like to thank Joe Halpern for pointing out several problems with an earlier version and Rimli Sengupta for discussions that helped in the development of this work.

References

[1] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[2] Ronald Fagin and Moshe Y. Vardi. Knowledge and implicit knowledge in a distributed environment: Preliminary report. In Joseph Y. Halpern, editor, *Proceedings of the First Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 187–206. Morgan-Kaufmann, March 1986. Also appears as Technical Report RJ4990, IBM Research Laboratory.

[3] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Department of Computer Science, Yale University, June 1983.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[6] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In J.-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes on Computer Science*, pages 110–123. Springer-Verlag, September 1989.

[7] Vassos Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols (preliminary report). In *Proceedings of the Sixth Symposium on Principles of Database Systems*, pages 129–134. ACM Press, March 1987.

[8] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[9] Joseph Y. Halpern, Yoram Moses, and Orli Waarts. A characterization of eventual Byzantine agreement. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 333–346, August 1990.

[10] Murray S. Mazer. A knowledge theoretic account of recovery in distributed systems: The case of negotiated commitment. In Moshe Y. Vardi, editor, *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 309–324. Morgan-Kaufmann, March 1988.

[11] Ruben Michel. *Knowledge in Distributed Byzantine Environments*. Ph.D. dissertation, Yale University, December 1989.

[12] Yoram Moses and Mark R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1):121–169, 1988.

[13] Gil Neiger and Rida Bazzi. Using knowledge to optimally achieve coordination in distributed systems. In Yoram Moses, editor, *Proceedings of the Fourth Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 43–59. Morgan-Kaufmann, March 1992.

[14] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[15] Gil Neiger and Mark R. Tuttle. Common knowledge and consistent simultaneous coordination. In J. van Leeuwen and N. Santoro, editors, *Proceedings of the Fourth International Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes on Computer Science*, pages 334–352. Springer-Verlag, September 1990. To appear in *Distributed Computing*.

[16] Mark R. Tuttle. A game-theoretic characterization of eventual common knowledge. Unpublished manuscript, October 1988.

From Sequential Layers to Distributed Processes

Deriving a Distributed Minimum Weight Spanning Tree Algorithm *

(EXTENDED ABSTRACT)

Wil Janssen & Job Zwiers
University of Twente †

Abstract

Analysis and design of distributed algorithms and protocols are difficult issues. An important cause for those difficulties is the fact that the *logical structure* of the solution is often invisible in the *actual implementation*. We introduce a framework that allows for a formal treatment of the design process, from an abstract initial design to an implementation tailored to specific architectures. A combination of algebraic and axiomatic techniques is used to verify correctness of the derivation steps. This is shown by deriving an implementation of a distributed minimum weight spanning tree algorithm in the style of [GHS].

1 Introduction

Protocols for distributed systems can not only be complicated to develop but even more complicated to understand by others than the designers. Such protocols are often the result of a process of transforming, refining and optimizing a basically simple algorithm. In order to explain and clarify the final resulting protocol, as opposed to mere verification, the structure of a correctness proof should reflect the structure of the original design. A notorious example is the algorithm for determining minimum weight spanning trees by Gallagher, Humblett and Spira [GHS]. There are several published correctness proofs of the [GHS] algorithm [WLL, CG, SR], some of which rely on a protocol structure for the verification process that differs from the structure of the final algorithm. Yet we feel that these proofs fall short of clarifying certain relevant aspects of the [GHS] algorithm. In this paper we identify such aspects and we show how each of them can be understood in a series of relatively easy transformations where at each step only a few new aspects are introduced. This leads to a natural decomposition of our correctness proof that has moreover the desirable property that it closely follows a (possible) design trajectory. Explanation in the form of systematic design allows for a comparison of algorithms by means of a "genealogy"; the earlier during the design that a different design decision was taken, the more different the finally resulting algorithms are. This genealogy often suggests other algorithms and improvements. For the [GHS] algorithm we present a design trajectory that starts with an initial solution from which algorithms can be obtained as divers as the Prim and Dijkstra algorithms [Pri, Dijk], Kruskal's algorithm [Kru], Boruvka's algorithm ([Bor, Tar]) and, indeed, the algorithm of [GHS]. Already at a very early stage in our design trajectory most of these, except Boruvka's, are excluded. We thus obtain a variant where essentially the time complexity claimed by [GHS] is achieved.

The transformational design that we propose goes from a *sequential program* (essentially Boruvka's algorithm) via a *sequentially phased parallel program* to a *distributed program*. A sequentially phased parallel program [SR] can be described as a sequential composition of a number of *layers* [EF, KP], each of which is a (relatively simple) parallel program. Many protocols for distributed systems admit such a "layered" presentation which is much easier to analyze than the final distributed version. In [JPZ] a formulation of this principle in the form of an *algebraic transformation law* has been put forward.

In the present paper we apply this transformation law and show that it can be applied to a situation as complex as the GHS protocol. We do so by systematically deriving a GHS-like protocol in a number of steps, starting with a simple sequential Boruvka-like algorithm, distributing it over nodes and introducing optimizations.

*Part of this work has been supported by Esprit/BRA Project 6021 (REACT)

†Address: University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands. E-mail: {janssenw,zwiers}@cs.utwente.nl

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0215...$1.50

This leads to a correctness proof in a number of relatively simple steps, reflecting decisions in the design process.

By several other authors it is also argued that the correctness proof should be able to represent the intuitive explanation given by the protocol designers. Chou and Gafni [CG] group classes of actions and define a sequential structure on such classes (so-called *stratification*). In the actual verification however they use singleton classes and a total order on actions which does not comply with the abstract protocol structure.
Stomp and de Roever on the other hand [SR] introduce what they call a *principle for sequentially phased reasoning* which allows them to introduce *semantically defined* layers that should correspond to the intuitive ideas of the designers. In [Sto] this principle is applied to the *derivation* of a broadcast protocol. The main difference to our approach that we use a formulation of this principle in an *algebraic* setting.

Both approaches are closely related to the idea of *Communication Closed Layers* by Elrad and Francez.

In order to get an idea of aspects of the GHS protocol that we can explain we provide some detail of the protocol as described in [GHS]. The protocol determines the minimum weight spanning tree (MST) of a given connected undirected graph with N nodes and E edges. A connected subgraph of the MST is called a *fragment*; virtually all algorithms for determining the MST start with trivial fragments in the form of single nodes and "grow" one or more fragments until the complete MST has been obtained. The basic principle to enlarge a fragment is to calculate its (uniquely determined) minimum-weight outgoing edge: this edge can be shown to be part of the MST. Two fragments can be combined by connecting them via edge e if e is the minimum weight outgoing edge of at least one of those fragments. In [GHS] each fragment finds its minimum-weight outgoing edge concurrently and asynchronously with regard to other fragments, and then tries to combine with the fragment at the the other end of the edge by sending a "connect" message. How and when this combination takes place is quite intricate and must be regarded as one of the typical characteristics of the GHS protocol. It depends on so called *levels* attached to fragments. Apart from single nodes which are defined to be at level 0, fragments F have a level $L > 0$ which, according to [GHS], depends on previous combinations. We quote [GHS]:

> "Suppose a given fragment F is at level $L \geq 0$ and the fragment F' at the other end of $F's$ minimum-weight outgoing edge is at level L'. If $L < L'$, then fragment F is immediately absorbed as part of fragment F', and the expanded fragment is at level L'. If $L = L'$ and fragments F and F' have the same minimum-weight outgoing edge, then the fragments combine immediately into a new fragment at level $L + 1$. In all other cases, fragment F simply waits until fragment F' reaches a high enough level for combination under the above rules."

One important reason why it is difficult to get a clear intuitive understanding of the protocol is that various fragments with totally different levels are active at the same time. Our analysis is based on a clear distinction between *causal order* and *temporal order* of events. It is shown that the apparent "chaos" from the temporal point of view corresponds to a highly regular pattern from a causal order point of view. Actually, in terms of causal order the protocol is closely related to Boruvka's algorithm, where there is a strict alternation between phases where the minimum weight outgoing edges of all fragments are determined and phases where fragments combine *until no further combination is possible anymore*. Conceptually, i.e. from a causal order point of view, *all* fragments of a given level L are created together, in a *single* phase as sketched. From a temporal point of view though, the creation of fragments, by means of creating a 'core' and followed by 'absorbing' other fragments, is spread out over time. Actually it is quite possible for a given fragment that it is already becoming part of some higher level fragment before the fragment itself is completed! Within this setting it now becomes possible to clarify an aspect such as the necessity of tagging messages with level numbers in [GHS]: In the intermediate stages of our design trajectory there are no such tags or any other (explicit) references to level numbers within the program itself. However, in order to apply our communication closed layers law we introduce separate sets of communication channels, one for each level so as to fulfill the side conditions for the law. After applying the law we have obtained a distributed algorithm with still the same sets of channels. In one of the last transformation steps we then *merge* the channels for different levels between two given nodes, by a straightforward *multiplexing* technique.

The framework we introduce in this paper allows to formulate principles like communication closed layers in a *compositional and algebraic setting*. The formulation of such laws strongly depends upon a new type of composition operator called *layer composition*. It is resembles sequential composition but allows more parallelism between actions. The definition of this operator is given in a partial order model, but does not depend on that. It relies on a symmetric and irreflexive relation between actions, called the *conflict relation*, akin to conflicts in distributed databases [BHG].

The introduction of such operators yield a language that allows us to follow the whole trajectory starting with an *initial design* that is free of architectural bias

to the actual *physical implementation* where considerations like the number of processors or nodes and the network structure. The derivation steps in this trajectory cannot be done by using algebraic laws only. At some moments in the derivation process *state-based* and *axiomatic* reasoning is needed to show the correctness. By combining both styles of reasoning only, we can bridge the gap between abstract specification and low-level implementation.

Before giving the derivation we first introduce the language used and the underlying model.

2 The design language and its properties

In this section we present the language used in the derivation process and some of the properties needed. In order to get some intuition for the validity of these laws and properties we informally describe the underlying model, based on runs consisting of a set of events and a *causal* order and a *temporal* ordering relation. For a more detailed treatment of the model and the algebraic properties of our language we refer to [JPZ, Zwi].

The language that we use is intended to be appropriate for the *initial design stage* – during which we prefer to have no bias towards a certain network architecture – as well as for the description of the final program that *should* fit the network structure. The reason for having a single language rather than two separate languages, one for initial design and another for coding the final program, is that we aim at a *gradual* transformation from initial design towards final implementation, which requires a single language that can represent all stages, including intermediate ones. Since we introduce some rather unconventional language operators, which are difficult to appreciate without a basic knowledge of the underlying model, we start with a sketch of the latter.

The model

Basically, we describe the execution of distributed systems by *histories* h that consist of a partially ordered set of *events*. This model is related to the *pomset* model as introduced in [Pratt]. Typical examples of events that we actually use in this paper include send and receive actions and read and write operations to local or shared memory. The precise interpretation of an event e is determined by its *attributes* $a(e)$, some of which will be mentioned below. For each system many different histories are possible, due to different behavior of the concurrent environment of the system and other causes of nondeterminism. Therefore a system *semantically* denotes a *set of possible histories*.

Events e and f that are *unordered* in some history h, are said to be independent. Potentially such events execute in parallel, i.e. at the same time or at overlapping time intervals. Within our design formalism there are two causes for *ordering* events which consequently do *not* execute in parallel:

- The first one is because e and f *conflict* in the sense that they both access a common resource that does not allow simultaneous access. The generic example (and the terminology) stems from conflicts between concurrent database actions [BHG] due to read and write operations to the same shared memory locations. When this happens e and f simply *cannot* execute (fully) in parallel and so must *logically* be ordered, which we denote as either as $e \longrightarrow f$ or as $f \longrightarrow e$, depending on which is the case. Only conflicting actions are ordered logically.
- The second cause is that actions are *temporally ordered* as the result of the use of language operators that explicitly require such ordering. Such operators are typically used in the last design stage where actions are actually *allocated* on specific processors, or to specific network nodes. Clearly, actions that should run on a single processors have to be ordered temporally. Temporal precedence of e over f is denoted by $e \longrightarrow\!\!\!\!\!\rightarrow f$.

Because of the sharp difference between logical and temporal precedence, conceptually from the point of view of a designer as well as from a more technical point of view, we use a formal semantic model where histories are structures of the form $(E, \longrightarrow, \longrightarrow\!\!\!\!\!\rightarrow)$, and where E is a set of events, with a dual ordering defined on it: $(E, \longrightarrow)$ is a directed acyclic graph (DAG), i.e., the transitive closure of $\longrightarrow$ is a *partial order* on E. $(E, \longrightarrow\!\!\!\!\!\rightarrow)$ is simply a partial order itself. The two ordering relations are weakly related. Temporal order obviously does not imply logical precedence. If two events e and f are *logically* ordered, say $e \longrightarrow f$, then they cannot be ordered *temporally* in the reversed direction, i.e.

$e \longrightarrow f$ implies $f \not\longrightarrow\!\!\!\!\!\rightarrow e$

Also the following relation must hold

$e \longrightarrow\!\!\!\!\!\rightarrow f \longrightarrow g \longrightarrow\!\!\!\!\!\rightarrow h$ implies $e \longrightarrow\!\!\!\!\!\rightarrow h$

Informally one can think of $e \longrightarrow f$ as e *influencing* f which cannot be the case if f completely precedes e in time. Any stronger relationship cannot be assumed; for instance from database serializability theory there are well known examples of atomic transactions e, f and g such that $e \longrightarrow f \longrightarrow g$, yet $g \longrightarrow\!\!\!\!\!\rightarrow e$!

Informal semantics and algebraic properties

The two main composition operators of the language, parallel composition and conflict composition, are de-

fined purely in terms of logical precedence, i.e. no temporal order is enforced by these operators.

The histories for a *parallel composed* system S of the form $Q \parallel R$ can be described as follows. The events executed by S in some history h can be partitioned into subhistories h_Q and h_R that are possible histories for Q and R. Moreover, the logical precedence relation *between* h_R and h_Q is such that all conflicting events are logically ordered, where the direction of the precedences are nondeterministically chosen. This nondeterminism is constrained of course by the fact that logical precedence is an *order*, so cycles of the form $e_0 \longrightarrow e_1 \longrightarrow \cdots \longrightarrow e_0$ are not allowed.

Layer composition can be considered an *asymmetric form of parallel composition*. For $Q \parallel R$ the logical precedence between conflicting actions of Q and R is nondeterministically determined. For layer composition of Q and R, which is denoted by $Q \bullet R$, actions from Q take logical precedence over actions from R in case of conflicts. In the case of independent actions no order is enforced however, just as is the case for parallel composition. We also use *iterated layer composition* $S^{\odot}$, analogously to iterated sequential composition $S^{\star}$.

Layer composition should be compared with *sequential composition* of the form $Q\,;R$. This is somewhat like layer composition except that we also enforce *temporal ordering* between Q and R actions: all Q actions temporally precede all R actions, regardless of conflicts. So whereas conflict composition admits parallel execution of certain actions, sequential composition does not. A sharp difference between the two forms of composition shows up when we consider Elrad and Francez' "communication closed layers" [EF]. The essence of communication closed layers is that under certain conditions a parallel system $S \parallel T$ where S and T are sequential programs of the form $S_0\,;S_1$ and $T_0\,;T_1$, is "equivalent" to a sequential composition of "layers" $S_0 \parallel T_0$ and $S_1 \parallel T_1$, thus:

$(S_0\,;S_1) \parallel (T_0\,;T_1) \equiv (S_0 \parallel T_0)\,;(S_1 \parallel T_1)(*)$

The side condition is that there is no communication, or in our parlance *no conflict*, between actions from S_0 and T_1, nor should there be conflicts between action from S_1 and T_0. Generalized forms of this principle appear also in [SR]. The equivalence used in $(*)$ is sometimes called IO-equivalence, referring to the fact that although the histories of left hand and right hand sides of $(*)$ are not the same, the relation between initial and final states of the system *is* the same nevertheless. A problem with this equivalence is that it is not a congruence, so we cannot simply interchange left and right hand side of $(*)$ *within contexts!* Within our framework we can replace the sequential composition in $(*)$ by conflict composition however, resulting in the following algebraic law given for the case of two layers consisting of two parallel components (with the same side conditions as for $(*)$).

$(S_0 \bullet S_1) \parallel (T_0 \bullet T_1) = (S_0 \parallel T_0) \bullet (S_1 \parallel T_1)$ (CCL)

Note that we not only have a congruence, but even semantic *equality* here, which is to be understood as the fact that both sides of the equation admit exactly the same partial order based histories. We also use a number of derived laws, see [Zwi]. A special case is the well-known independence law:

If P and Q are non-conflicting, then

$P \bullet Q = P \parallel Q$

The process term $\mathbf{io}(P)$ denotes execution of a single action that captures the net effect of executing P without admitting interference by other events. The $\mathbf{io}(\cdot)$ operation is also called the *contraction* operation, since it contracts complete P runs into single events. Intuitively $\mathbf{io}(P)$ represents the input-output behavior of a process P if we execute that process in isolation, i.e. without interference from outside. This operation induces an interesting process equivalence, called IO-equivalence, and an associated IO-refinement relation. Such equivalences play an important role in the book by Apt and Olderog [AO].

$P \stackrel{IO}{=} Q$ iff $\mathbf{io}(P) = \mathbf{io}(Q)$

Specification of what is often called the *functional behavior* of a process P is really a specification of $\mathbf{io}(P)$, i.e. of the IO-equivalence class of P. The $\mathbf{io}(\cdot)$ operation does (obviously) not distribute though parallel composition. For the case of layer composition we have the following law:

$P \bullet Q \stackrel{IO}{=} \mathbf{io}(P) \bullet \mathbf{io}(Q)$

The intuition here is that although execution of "layer" P might overlap execution of "layer" Q *temporally*, one can pretend that all of P, here represented as an atomic action $\mathbf{io}(P)$, precedes all of Q as far as IO-behavior is concerned.

IO-behavior of a system can also be *specified* by means of classical pre- and postconditions. We interpret a Hoare style formula of the form:

$\{pre\}\ S\ \{post\}(**)$

where *pre* and *post* are *state formulae* as usual, as follows. For each history h in $\mathbf{io}(S)$ let $s_0(h)$ and $s(h)$ denote the initial and final state of the (unique) S event in h. Then $(**)$ requires that if the initial state $s_0(h)$ satisfies precondition *pre* the corresponding final state $s(h)$ satisfies the postcondition *post*. Hoare style program verification for concurrent systems is more complicated than verification of sequential programs due to the possibility of interference. The classical proof system for shared variables by Owicki and Gries [OG] for instance includes extra interference freedom checks for assertions used in proof outlines. It has been shown by Apt and Olderog [AO] that for *restricted cases* it is possible to verify parallel programs relying on techniques for *sequential* programs however. This work relies on classical Hoare style verification in combination with

program transformation based on IO-equivalence. We use similar techniques in the derivation of the algorithm, where we exploit the fact that conflict composition, although it does admit parallelism, behaves just like sequential composition when we apply the **io**() operation! This follows from the fact that the contraction of some history h can be determined without taking temporal ordering into account; logical precedence as such is sufficient to determine the cumulative state transformation associated with h.

This implies that to verify a pre-post specification for a program of the form $S \bullet T$ it suffices to verify the associated *sequential* program $S\,;T$.

In the derivation we use the combination of Hoare style formulas and program transformation to guarantee the correctness of some transformation steps. This can be seen as *proof outline transformation*, in the style of Reynolds ([Rey].) For example we have the following rule for iterated layer composition.

Define **layer** l : $P(l)$ **until** B as $(P \bullet B)^{\odot} \bullet \neg B$. If $P(l)$ is of the form

$P(l) \stackrel{\Delta}{=}$ **for** $v \in V$ **dopar** $P(v)(l)$ **rof** ,

and B is of the form

$B \stackrel{\Delta}{=} \forall v \in V(B(v))$,

and if furthermore the following premis are satisfied. For $l \neq l', v \neq v'$:

$P(v)(l)$ does not conflict with $P(v')(l')$

and

$\{B\}\ P\ \{B \vee (\forall v \in V(\neg B(v))\}$

then

layer l
for $v \in V$ **dopar** $P(v)(l)$ **rof**
until B

= { CCL- iteration }

for $v \in V$ **dopar**
layer l : $P(v)(l)$ **until** B
rof

Informally the last premisse states that all parallel components must stop at the same number of iterations.

We conclude this section with a somewhat more detailed description of the shared memory model and the communication mechanism used in the description of the algorithm.

Shared memory and communication

In our model the basic actions are *guarded assignments* of the form

$b\&x_1, x_2, \ldots, x_m := exp_1, exp_2, \ldots, exp_m$

Informally such an assignment is postponed until the guard b holds, where after the values of the expressions exp_i are assigned simultaneously to all x_i. So our guarded assignments are really limited forms of the well known **await** statement. If the guard is *true* it is omitted.

We assume that there exists a given *conflict relation* between actions, for example two action conflict if one writes into a variable the other action reads or writes. We could also assume read-read conflict too, but will not do so in this paper. At later stages we also use communication via channels. We can model *undirectional, asynchronous* channels by shared variables. A channel c can be defined as a pair $(c.flag, c.val)$ where $c.flag$ is a boolean that is true iff a value is available on the channel, and $c.val$ the value to be read. Send and receive actions can now be modeled as guarded assignments. The *channel name* c of send and receive actions is a triple given by the node the emitting the message, the node receiving it, and a name. Let $c = (u, v, \text{MSG})$:

$send(u)(v)(\text{MSG}(e)) \stackrel{\text{def}}{=}$
$\neg c.flag\&\ c.flag, c.val := true, e$

$receive(u)(v)(\text{MSG}(x)) \stackrel{\text{def}}{=}$
$c.flag\&\ c.flag, x := false, c.val$

We can take a more liberal view, where we have buffered channels, which is needed in the final stages of the derivation. We do not present a full syntax of the language used in the derivation. The operators used are straightforward abbreviations of expressions using the operators given above.

3 Derivation of the algorithm

As we explained in the introduction, the derivation follows a number phases, starting of with a simple and easy to prove sequential program and finally arriving at a distributed and partially optimized set of processes. In this section we give an outline of the total derivation process and exemplify a number of crucial steps in the development. The derivation is presented as a top-down structured process. This does not comply with the true derivation process as both the initial design and the final implementation were known on beforehand. The derivation given is the result of closing the gap from both sides, eventually resulting a clear derivation showing the correctness of the distributed implementation. The final result of the derivation follows the GHS protocol closely, but has some improvements from the point of view of top-down design of programs. Furthermore not all optimizations are introduced. See [JZa] for a derivation of the whole protocol.

In the derivation we can distinguish a number of different stages, each given by a number of relatively simple transformation steps:

1. The initial (sequentially structured) design
2. Distributing data
3. Recursively computing the minimum weight outgoing edge

4. Synchronization and information diffusion by means of message passing
5. Applying the Communication Closed Layers law to get a distributed implementation
6. Multiplexing channels
7. Optimizing the algorithm

Of all these steps, the application of CCL is a purely algebraic one, although it requires some non-algebraic transformations in order to satisfy the premisses of CCL. Other parts of the derivation are proven in an axiomatic way or as a combination of both strategies. We will emphasize the first few steps and the application of the CCL laws.

The initial design closely follows the algorithm introduced by Boruvka which can be found in [Tar]. As it is a *sequentially structured* algorithm its correctness can be shown using classical Hoare style techniques [Lam]. It is not purely *sequential* however as it is formulated using layer composition instead of sequential composition. This to allow the program to be transformed and distributed using the algebraic framework we introduced. However, the overall behavior of this system can be viewed as if it executes sequentially. According to our viewpoint the use of sequential composition should be restricted to those cases when one really means to introduce a *temporal* relation instead of a causal relationship. In an initial design this is hardly ever the case, as no architectural decisions have been taken into account yet.

Before describing this algorithm we introduce some notation and theorems on minimum weight spanning trees.

3.1 Spanning trees and fragments

Assume we have a given connected and undirected graph $G = (V, E)$. We assume every edge i has a distinct weight $w(i)$. We assume all nodes have distinct names and are totally ordered. In the following let u, v, x and y denote vertices, and let $i, j, k, \ldots$ denote edges. Edges are also denoted by two-element sets $\{u, v\}$. For a graph G the following theorem holds

Theorem 3.1
If G is a connected graph where every edge has a distinct weight, the minimum weight spanning tree $MST(G)$ is uniquely determined. □

The proof can be found in [GHS].

For any node $v \in V$ let $inc(v)$ denote the set of edges incident to v, i.e.

$inc(v) \stackrel{\text{def}}{=} \{ \{v, u\} \in E\}$

For an edge $j = \{u, v\}$ let the *destination* of j with respect to u, $dest(u)(j)$ be v. We also use the source or destination of an edge w.r.t. a fragment or set of nodes, e.g. for fragment f and edge $j = \{u, v\}$ such that $v \notin f$ and $u \in f$ we have that $src(f)(j) = u$.

A *fragment* is a connected subgraph of $MST(G)$. For any fragment f let $\mu(f)$ be the minimum weight outgoing edge of f.

The basic idea of the algorithm follows from the following lemma, which is proven in [GHS].

Lemma 3.2
Let $G = (V, E)$ be a connected graph where every edge has a distinct weight, and let f be a fragment of $MST(G)$. If k is the minimum weight outgoing edge of f then joining k and its adjacent nonfragment node to f yields another fragment of $MST(G)$. □

In the same way we can combine two *fragments* with a connecting minimum weight outgoing edge into a new fragment.

The algorithm we introduce in the next section is based on Boruvka's algorithm [Bor, Tar]. The rough idea is as follows. We compute a set of fragments *frag* by iteratively combining fragments and their minimum weight outgoing edges. Initially *frag* is the set of all fragments that consist of a single node and no edges (which is a fragment by definition). Then every fragment determines it minimum weight outgoing edge and combines with the fragment on the other side of the edge. If two fragments share the same minimum weight outgoing edge j, then j is called the *core* of the newly formed fragment. The node adjacent to the core with the least name is called the *core node*. [1] If a fragment is not combined via a core to another fragment it is said to be *absorbed*.

The algorithm terminates when we only have a single fragment left, $MST(G)$.

Every fragment in this algorithm consists of a *core node* that is the root of the tree consisting of all other branches and other nodes in the fragment. This tree structure is used to gather information in the tree or to broadcast information.

In the derivation we also need the following lemma. One of the characteristic features of the GHS protocol – postponement of absorption – is partially based on this lemma.

Lemma 3.3
Let f be a fragment and let j be an edge of f. Removing j (but not its endpoints) from f disconnects f in two disjoint trees, at least one of which – say f_1 – does *not* contain the core of f. (if j is the fragment core any of the two subtrees can be taken.) We then have that $\mu(f_1) = j$.

Proof: see [JZa] □

[1] This is different from [GHS] where both nodes adjacent to a core play equivalent roles. In the top-down design of the algorithm the choice for a single core node is more straightforward and leads to more elegant solutions, without essentially changing the ideas of GHS. We therefore take this choice.

In the rest of this paper we furthermore use the following operations on graphs and trees. Let $G = (V, E)$ and $H = (V', E')$ be graphs. We now define the union of G and H as

$G \cup H \stackrel{\text{def}}{=} (V \cup V', E \cup E')$

For node v and edge k let

$v \in G \stackrel{\text{def}}{=} v \in V$ and $k \in G \stackrel{\text{def}}{=} k \in E$,

and

$G \cup \{v\} \stackrel{\text{def}}{=} (V \cup \{v\}, E)$,

$G \cup \{k\} \stackrel{\text{def}}{=} (V, E \cup \{k\})$.

Furthermore, for any fragment f let $inc(f)$ be the set of edges leaving f (i.e. the set of all $\{u, v\}$ such that $u \in f$ and $v \notin f$). For a node u we define $out(f)(u)$ as the set of edges incident to u leaving f, i.e.

$out(f)(v) \stackrel{\text{def}}{=} inc(v) \cap inc(f)$

If f is clear from the context it is omitted.

We define the minimum weight edge of a non-empty set E, $min\_edge(E)$ as

$min\_edge(E) \stackrel{\text{def}}{=} e \in E$ such that
$w(e) = MIN\{w(e') \mid e' \in E\}$

and define $min\_edge(\emptyset) \stackrel{\text{def}}{=} nil$. We take $w(nil) \stackrel{\text{def}}{=} \infty$. Finally, for a graph $G = (V, E)$, let $concomp(G)$ be the set of connected componenents in G, i.e. the set of maximal and connected graphs in G.

3.2 The initial design

The first implementation is based on the construction of a set of fragments *frag* which determine their minimum weight outgoing edges and combine connected fragments. Initially the set *frag* consists of all trees consisting of a single node and no edges, which are by definition fragments. Furthermore we compute the set of edges B that are branches, i.e. that are part of the spanning tree. This is the basic idea of Boruvka's algorithm. It can be described as:

$MST_0 \stackrel{\wedge}{=}$
  $B := \emptyset \bullet$
  $frag := concomp((V, B)) \bullet$
  layer
    $M := \{min\_edge(inc(f)) \mid f \in frag, inc(f) \neq \emptyset\} \bullet$
    $B := B \cup M \bullet$
    $frag := concomp((V, B))$
  until $M = \emptyset$

The total correctness of this algorithm follows from loop invariant I_0, the definition of $\mu(f)$, and bound function τ, that are defined as:

I_0 : $|frag| \geq 1 \wedge$
  $\forall f \in frag(f$ is a subtree of $MST(G)) \wedge$
  $\{V' \mid (V', E') \in frag\}$ is a partitioning of V

and

$\tau \stackrel{\text{def}}{=} \log(|frag|)$

The invariance follows from the initialization and lemma 3.2. The number of fragments $|frag|$ is at least divided by two in each iteration and therefore $\log(|frag|)$ decreases. From the invariant and the termination condition the postcondition

P_0 : $frag = \{MST(G)\}$

easily follows.

Although we take MST_0 as the initial design in our trajectory, it is also possible to give an even more general algorithm that comprises the Prim-Dijkstra and Kruskal algorithms, by not adding M to B, but only adding a subset of M to B. In that case however we loose the logarithmic complexity of the algorithm, as the number of fragments decreases, but is not necessarily divided by two.

As a second step we split the computation of M into the layered computation of the minimum weight outgoing edge for every fragment. The reason for doing so is that we want to distribute data. Firstly per fragment, eventually per node. We do so by introducing a variable $mo(f)$ for every fragment f. This is an instance of straightforward top-down design and Hoare style verification for sequential programs. The following representation function for M will hold:

$M(mo) \stackrel{\text{def}}{=} \{mo(f) \mid f \in frag, inc(f) \neq \emptyset\}$

This leads to the following refined program:

$MST_1 \stackrel{\wedge}{=}$
  $B := \emptyset \bullet$
  $frag := concomp((V, B)) \bullet$
  layer
    for $f \in frag$ **layer**
      if $inc(f) \neq \emptyset$ **then**
        $mo(f) := min\_edge(inc(f)) \bullet$
        $B := B \cup mo(f)$
      else $mo(f) := nil$
      fi
    rof $\bullet$
    $frag := concomp((V, B))$
  until $\bigwedge\{mo(f) = nil \mid f \in frag\}$

The correctness of this transformation step can be proven by means of the representation function M and the structure of the conditional statement that implies that

$mo(f) = nil$ iff $inc(f) = \emptyset$

as $inc(f) \neq \emptyset$ implies $min\_edge(inc(f)) \neq nil$.

3.3 Distributing data

The transformation of MST_1 to a program where all data are distributed takes a number of steps. First we will distribute B by introducing variables SE for every edge. This allows for the introduction of parallelism between the different fragments as conflicts due to the acces to B are resolved.

Thereafter we introduce variables $lmo(v)$ giving the local minimum weight outgoing edges of every node of ev-

ery fragment. Finally we introduce a core node for every fragment, by giving defining boolean variables $core(v)$ for every node v, such that for every fragment there is a single node in the fragment with $core(v)$.

We first introduce variables

$SE(u)(v) \in \{branch, basic\}$

for every $\{u, v\} \in E$. The following representation function $B(SE)$ will hold:

$B(SE) \stackrel{\text{def}}{=} \{\{u, v\} \in E \mid SE(u)(v) = branch\}$

The transformation consists of adding initialization of every $SE(u)(v)$ and of replacing

$B := B \cup \{mo(f)\}$

by

$SE(src(f)(mo(f)))\,(dest(f)(mo(f))) := branch$

The guard $inc(f) \neq \emptyset$ can also be replaced by $mo(f) \neq nil$ after computing $mo(f)$ as $min\_edge(\emptyset) = nil$.

The correctness of this step can be proven by VDM style data refinement with atomicity constraints. These constraints are fulfilled as every layer is interference free, because all actions are placed in the same layer and no parallel interfering processes exist.

After this transformation we can replace the layered construct **for** $f \in frag$ **layer** by **for** $f \in frag$ **dopar** , as all conflict are resolved. The correctness of this transformation is guaranteed by the independence law.

The code of the resulting program is omitted.

Now we introduce local minimum weight outgoing edges for every node in every fragment, $lmo(v)$. For these variables the following invariant must hold.

$I_1 : mo(f) = min\_edge(\{lmo(f) \mid f \in frag\})$

This, plus the previous changes, leads to the following algorithm:

$MST_2 \stackrel{\Delta}{=}$
  $Init$•
  layer
    for $f \in frag$ **dopar**
      for $v \in f$ **dopar**
        $lmo(v) := min\_edge(out(f)(v))$
      rof •
      $mo(f) := min\_edge(\{lmo(v) \mid v \in f\})$•
      if $mo(f) \neq nil$ **then**
        $SE(src(f)(mo(f)))\,(dest(f)(mo(f))) := branch$
      fi
    rof •
    $frag := concomp((V, B(SE)))$
  until $\bigwedge\{mo(f) = nil \mid f \in frag\}$

where
  $Init \stackrel{\Delta}{=}$
    for $v \in V$ **dopar**
      for $\{u, v\} \in inc(v)$ **dopar**
        $SE(u)(v) := basic$
      rof
    rof •
    $f := concomp((V, B(SE)))$

The correctness of these transformation steps can again be easily verified (see [JZa]).

In MST_2 we still have the set of fragments $frag$ as a variable. This information must be localized too. We do so by introducing boolean variables $core(v)$ for every node v, and defining a function $\mathcal{F}rag(u)$ giving the fragment u belongs to, i.e. the set of nodes and edges connected to u by branches.

$\mathcal{F}_b(v) \stackrel{\text{def}}{=} \{u \in V \mid connected(v, u)\}$

$\mathcal{F}rag(v) \stackrel{\text{def}}{=}$
  $(\mathcal{F}_b(v), \{\{u, u'\} \in \mathcal{F}_b(v)^2 \mid SE(u)(u') = branch\})$

where *connected* means that v and u are connected via a path of branches. We leave its definition implicit.

We define the following representation function and (data) invariants:

$F(core) \stackrel{\text{def}}{=} \{\mathcal{F}rag(u) \mid u \in V, core(u)\}$

$MO(\mathcal{F}rag(u)) \stackrel{\text{def}}{=} mo(v)$ such that $v \in \mathcal{F}rag(u) \wedge core($

$I_2 : \forall v \in V\,(\exists! u \in \mathcal{F}rag(v)\,(core(u)))$

$I_3 : \forall \{u, v\} \in E\,(SE(u)(v) = SE(v)(u))$

Data invariant I_3 is now needed to guarantee the correctness of the definition of *connected*. Furthermore we will make use of this in later stages.

We define the set $Core$ as

$Core \stackrel{\text{def}}{=} \{v \in V \mid core(v)\}$

How can we now achieve I_2, i.e. how do we choose a unique core node for every fragment? Initially this is no problem: every fragment consists of a single node. After that we know that for every new fragment there were two exactly subfragments that had the same minimum weight outgoing edge, the core edge. As all nodes are ordered we can take the first of the nodes adjacent to the core. In order to determine which nodes are adjacent to minimum weight outgoing edges we introduce variables $con\_req(u)(v)$ stating that $\{u, v\}$ was the minimum weight outgoing edge of $\mathcal{F}rag(u)$. The following invariant will therefore hold:

$I_4 : \forall u \in Core\,(con\_req(v)(v')$ iff
  $\{v, v'\} = mo(u) \wedge v \in \mathcal{F}rag(u))$

Apart from some minor changes in MST_1 we have to establish I_4 at the end of the loop, i.e. the final statement in the **layer** ... **until** will be *ComputeCore*,

defined as

ComputeCore $\triangleq$
  for $u \in Core$ **dopar** $core(u) := false$ **rof** •
  for $v \in V$ **dopar**
    for $\{v, x\} \in inc(v)$ **dopar**
      if $con\_req(v)(x)$ **then**
        $con\_req(v)(x) := false$ •
        if $SE(v)(x) = branch$ **then**
          if $v < x$ **then**
            $core(v) := true$
          fi
        else $SE(v)(x) := branch$
        fi
      fi
    rof
  rof

The correctness of this transformed MST_2, which we will call MST_3, can again be proven by proof outline transformation and Owicki-Gries style verification with trivialized interference freedom tests because of local variables. In this proof lemma 3.2 and theorem 3.1 are needed. The proof itself is omitted.

3.4 Recursively computing the minimum weight outgoing edge

Now we have introduce core nodes we can make use of the fact that we can view a fragment as a rooted tree to compute $mo(u)$ for every core node u. To do so we introduce variables $up(v)$ denoting the edge toward the root of node v in the fragment (for $\neg core(v)$). Furthermore we introduce a variable $mo(v)$ for every node v, not only core nodes, denoting the minimum weight outgoing edge of the tree in the fragment of which v is the root.

We also need to *synchronize* the nodes in a fragment: a node v needs the values of all its successors in the tree to compute $mo(v)$. For this purpose we define synchronization flags $mo\_comp(u)$ that are set to true when the value of $mo(u)$ has been computed. Synchronization is also needed to inform edges that have to change their root and upward edge, i.e. nodes that are on the path from the core to the minimum weight outgoing edge, as we do not want to implement this by core actions only. For this purpose we introduce three-valued flags $change(u) \in \{true, false, \bot\}$. In the algorithm below these synchronization variables are indexed by the number of the layer. This in order to guarantee synchronization w.r.t. to *that* layer, or, viewed differently, to make the layers communication closed. We come back to that later.

The following notation is used:

$down(v) \stackrel{def}{=} \{j = \{v, u\} \in inc(v) \mid SE(v)(u) = branch \wedge j \neq up(v)\}$,

$tree(v) \stackrel{def}{=} \{(\{v\}, down(v))\} \cup \bigcup\{tree(u) \mid \{v, u\} \in down(v)\}$,

and

$$root(v) \stackrel{def}{=} \begin{cases} v, & \text{iff } core(v) \\ root(dest(v)(up(v))), & \text{iff } \neg core(v) \end{cases}$$

We furthermore define the path between two (connected) nodes u and v as the sequence of nodes on the path. The full definition is omitted. For a path $p = [uvw\ldots]$ we define the first edge of p, $first(p)$, as the pair $\{u, v\}$.

For the next algorithm MST_4 the following invariants will hold:

I_5 : $mo\_comp(v) \Rightarrow mo(v) = min\_edge\{mo(u) \mid u \in tree(v)\}$

I_6 : $(mo\_comp(v) \wedge down(v) = \emptyset) \Rightarrow mo(v) = lmo(v)$

I_7 : $\forall u, v \in V((core(v) \wedge connected(u, v)) \Rightarrow up(u) = first(path(u, v)))$

For sake of brevity we immediately introduce a second addition in this algorithm. Let $be(v)$ be the edge leading to the minimum weight outgoing edge, i.e.

I_8 : $\forall u, v \in V((u \neq v \wedge lmo(u) = mo(v) \wedge connected(v, u)) \Rightarrow be(v) = first(path(v, u)))$

I_9 : $lmo(u) = mo(u) \Rightarrow be(u) = lmo(u)$

All this leads to the following algorithm:

MST$_4$ $\triangleq$
  Init •
  layer l
    ComputeLocal(l) •
    ComputeGlobal(l) •
    ChangeRootPath(l) •
    ComputeCore(l)
  until $\bigwedge\{mo(v) = nil \mid v \in V\}$

where

Init $\triangleq$
  for $u \in V$ **dopar**
    $core(u) := true \| up(v) := nil \| be(v) := nil \|$
    for $\{u, v\} \in inc(u)$ **dopar**
      $SE(u)(v) := basic \| con\_req(u)(v) := false$
    rof
  rof ,

ComputeLocal(l) $\triangleq$
  for $u \in Core$ **dopar**
    for $v \in \mathcal{F}rag(u)$ **dopar**
      $lmo(v) := min\_edge(out(\mathcal{F}rag(u))(v)) \|$
      $mo\_comp(v)(l) := false$ **rof**
  rof ,

```
ComputeGlobal(l) ≙
  for u ∈ Core dopar
    for v ∈ Frag(u) dopar
      mo(v), be(v) := lmo(v), lmo(v)
      for {v, x} ∈ down(v) dopar
        await mo_comp(x)(l) do
          if w(mo(x)) < w(mo(v)) then
            mo(v), be(v) := mo(x), {v, x}
          fi
        od
      rof ;
      mo_comp(v)(l) := true
    rof
  rof
```

Note that we need the sequential composition at this stage to enforce the right moment of synchronization. Furthermore let

```
ChangeRootPath(l) ≙
  for u ∈ Core dopar
    change(u)(l) := (mo(u) ≠ nil) •
    for v ∈ Frag(u) dopar
      await change(v) ≠ ⊥ •
      if change(v)(l) then
        up(v) := be(v) •
        if SE(v)(dest(v)(be(v)) = branch then
          change(dest(v)(be(v)))(l) := true
        else
          SE(v)(dest(v)(be(v)) := branch •
          con_req(v)((dest(v)(be(v)) := true
        fi
      fi •
      for i ∈ down(v) − {be(v)} dopar
        change(dest(v)(i))(l) := false
      rof
    rof
  rof
```

and $ComputeCore(l)$ analogously to MST_3.

The correctness of the above solution is quite involved as it includes proving deadlock freedom and correctness of the recursive definition. [2] The proof however can be restricted to a single layer within a single execution of the loop which simplifies matters to a large extend. It can be proven correct using Hoare logic [Lam] or temporal logic [MP].

3.5 Introducing message passing

In MST_4 we had to introduce variables to synchronize actions and we had to copy values computed. As we are thriving for a distributed solution it is very well possible to introduce *communication over channels* to enforce synchronization and to pass values. As send and receive actions are defined as guarded assignments this transformation is straightforward, and simplifies matters to a large extend.

Furthermore we want to remove all shared accesses from the algorithm, as these are not possible in distributed implementations. We therefore have to adapt the computation of *lmo*, remove the shared accesses to *con_req*, $mo(x)$, and references to $Frag(u)$. This is done by introducing message passing and variables $fn(v)$ denoting the fragment name of v.

Some further simplifications are possible: we hardly ever refer to the edge $mo(v)$, but often to its weight. We therefore use variables *bw* instead of *mo*. Also the variables *lmo* are oblivious as their function can be taken by *bw*. Finally we can join the parallel executions over all core nodes and all nodes in the corresponding fragment to the parallel execution over every node in V.

The result of these transformations, MST_5 has the following structure. The full code is omitted because of space limitations.

```
MST5 ≙
  for v ∈ V dopar Init(v)' rof •
  layer l
    for v ∈ V dopar ComputeLocal(v)(l)' rof •
    for v ∈ V dopar ComputeGlobal(v)(l)' rof •
    for v ∈ V dopar ChangeRoctPath(v)(l)' rof •
    for v ∈ V dopar ComputeCore(v)(l)' rof •
    for v ∈ V dopar ChangeName(v)(l)' rof
  until ⋀{bw(v) = ∞ | v ∈ V}
```

The processes $ComputeLocal'$ and $ComputeCore'$ can both be split into two processes by means of algebraic transformations and proof outline transformations. The former can be split into a kernel process concerned with computing $be(v)$ and a test handler $TH(v)$ reacting upon TEST messages sent by other processes, the latter into a process possibly trying to connect and a connect handler $CH(v)$ responding to CONNECT messages of other nodes.

In this process we use the rule that if there are only a single send and a single receive action on a channel, executing them in parallel is the same as first sending and then receiving (see [JZ]).

We define $basic(v) \stackrel{def}{=} \{\{v, x\} \in inc(v) \mid SE(v)(x) = basic\}$. The processes $TH(v)$ and $CH(v)$ have the following form:

```
TH(v) ≙
  for {v, x} ∈ basic(v) dopar
    receive(v)(x)(TEST(fn(x)(v))) •
    if fn(x)(v) = fn(v) then send(v)(x)(REJECT)
    else send(v)(x)(ACCEPT)
    fi
  rof
```

[2] The solution given above deviates from [GHS] in the fact that we only change $up(v)$ on the path from the core to the new minimum weight outgoing edge. In [GHS] every $up(v)$ variable is reset in every iteration when an INITIATE is received.

$CH(v) \triangleq$
for $\{v, x\} \in basic(v)$ **dopar**
$send(v)(x)(\text{NoConnect})\|$
$(receive(v)(x)(\text{NoConnect})$ **or**
$(receive(v)(x)(\text{Connect}) \bullet SE(v)(x) := branch))$
rof

3.6 Applying Communication Closed Layers

What we eventually want to arrive at is a distributed implementation, i.e. an implementation of the form

$MST_f \triangleq$
for $v \in V$ **dopar**
$Init(v) \bullet$
layer $P(v)$ **until** $B(v)$
rof

The algorithm MST_5 however still is of a sequential nature. We want to apply the Communication Closed Layers Law to transform MST_5 to a distributed structure. To be able to apply the CCL law or its iterated version MST_5 must be of the correct structure and fulfill the premisses.

In order to arrive at the structure desired we first have to transform the loop body. This consists of a given number of layers that are all of the form

$L_i \triangleq$ **for** $v \in V$ **dopar** $L_i(v)$ **rof**

This is the correct structure to apply CCL. Furthermore all layers are communication closed as communication takes place within a layer, and other conflicts only exists within the process of a single node. We can transform the loop body L now as follows:

L
$=$ { by definition }
$(ComputeLocal' \| TH) \bullet$
$ComputeGlobal' \bullet$
$ChangeRootPath' \bullet$
$(ComputeCore' \| CH) \bullet$
$ChangeName'$
$=$ { $\|$ is commutative and associative }
for $v \in V$ **dopar** $ComputeLocal(v)' \| TH(v)$ **rof** $\bullet$
for $v \in V$ **dopar** $ComputeGlobal(v)'$ **rof** $\bullet$
for $v \in V$ **dopar** $ChangeRootPath(v)'$ **rof** $\bullet$
for $v \in V$ **dopar** $ComputeCore(v)' \| CH(v)$ **rof** $\bullet$
for $v \in V$ **dopar** $ChangeName(v)'$ **rof**
$=$ { CCL }
for $v \in V$ **dopar**
$(ComputeLocal(v)' \| TH(v)) \bullet$
$ComputeGlobal(v)' \bullet ChangeRootPath(v)' \bullet$
$(ComputeCore(v)' \| CH(v)) \bullet$
$ChangeName(v)'$
rof
$=$ { by definition $P(v)$ }
for $v \in V$ **dopar** $P(v)$ **rof**
$= L'$

We have now transformed L to a form suitable for the application of CCL. The guard of the loop however do not satisfy the premis of the iterated CCL rule:

$\{B\}\ P\ \{B \vee (\forall v \in V(\neg B(v))\}$

The last layer of the loop however consists of a broadcast of the name of the fragment. We change *ChangeName* in such a way that the new fragment name is *term*. This allows us to restate the termination condition as:

$B' \triangleq \bigwedge\{fn(v) = term \mid v \in V\}$

This condition does satisfy the premis, as

$\exists v \in V(fn(v) = term) \Rightarrow \forall v \in V(bw(v) = \infty)$

We can now transform MST_5 as follows:

MST_5
$=$ { by definition }
$Init' \bullet$ **layer** $l : L$ **until** B'
$=$ { $L = L'$ }
$Init' \bullet$ **layer** $l : L'$ **until** B'
$=$ { definition L' }
$Init' \bullet$
layer l
for $v \in V$ **dopar** $L(v)'$ **rof**
until B'
$=$ { iterated CCL }
$Init' \bullet$
for $v \in V$ **dopar**
layer $l : P(v)$ **until** $fn(v) = term$
rof
$=$ { CCL }
for $v \in V$ **dopar**
$Init(v)' \bullet$ **layer** $l : P(v)$ **until** $fn(v) = term$
rof
$=$ { by definition }
for $v \in V$ **dopar** $N(v)$ **rof**
$= MST_6$

The result of these transformations MST_6 has the desired distributed structure.

3.7 Multiplexing channels and optimizing messages

In MST_6 we used different channels for different layers. This asssumption is not realistic but it is possible to multiplex a number of channels on a single (buffered) channel. The buffer length must be at least the number of layers that are executed, which is limited by

$\log(|V|)$. A channel c is in this case not given by a pair $(flag, val)$, but by a function $c : [0..l] \rightarrow (flag, val)$. We can now assume that the level number is tagged to every message and we can implement a send action $send(v)(u)(\text{MSG}(e, ln))$ as

$send(v)(u)(\text{MSG}(e, ln)) \triangleq$
$(\neg c(ln).flag)\&c(ln).flag, c(ln).val := true, e$

By now we have transformed the algorithm in MST_6 where $N(v)$ has a structure as in fig. 1, where some subroutines have been left implicit. In this algorithm a number of optimizations are possible. First of all we can group statements for core and non-core nodes by using proof outlines and transforming them. By introducing the right invariants we can furthermore show that some messages (e.g. NOCONNECT and NOCHANGE) are oblivious and can be removed.

There are also some other optimizations possible w.r.t. to the number of messages: if we received a REJECT message on some basic edge, we will always receive a REJECT message. Introducing a *reject* status for edges will save double status requests. We can also check basic edges one by one, instead of all in parallel. In that case we have to check them in order of weight. This optimization allows us to postpone the absorption of fragments: if the edges allow which the fragment sent a CONNECT has to large a weight lemma 3.3 guarantees that we can absorb it at a later stage.

The details of these optimizations can be found in the full paper ([JZa]).

The final implementation step is now to replace *layer composition* by *sequential composition*, and to replace *parallel composition* by *sequential iteration* within a component. This does not invalidate the correctness (for non-interfering parallelism) and results in an *implementable, distributed algorithm.*

4 Conclusion

The layering techniques used to derive the implementation of a distributed minimum weight spanning tree algorithm have proven to be a powerful means in the development of parallel systems. This also holds for a posteriori verification where it can give insight in the structure of the implementation and the intuitive ideas of the designers.

These techniques are applicable to a large number of problems, not only to this type of algorithms. Other examples – varying from parsing algorithms to database protocols – can be found in [JZ], [JPZ], and [PZ].

At this moment we are investigating the relation between the process based approach as used in this paper and logic based approaches to layering like [KP]. The use of non-static dependency relations might be useful in our context too.

Also algorithms relying on real-time synchronization

$N(v) \triangleq$
$up(v) := nil \| core(v) := true \| fn(v) := v \|$
for $\{v, x\} \in inc(v)$ **dopar**
$se(v)(x) := basic \bullet con\_req(v)(x) := false$
rof $\bullet$
layer l
$be(v), bw(v) := nil, \infty$
(**for** $\{v, x\} \in basic(v)$ **dopar**
$send(v)(x)(\text{TEST}(fn(v), l)) \bullet$
$(receive(v)(x)(\text{REJECT}(l))$ **or**
$(receive(v)(x)(\text{ACCEPT}(l)) \bullet$
$\langle$ **if** $w(\{v, x\}) < be(v)$ **then**
$bw(v), be(v) := w(\{v, x\}), \{v, x\}$
fi $\rangle))$
rof $\| TH(v)) \bullet$
for $\{v, x\} \in down(v)$ **dopar**
$\langle receive(v)(x)(\text{REPORT}(b(v), l) \bullet$
if $b(v) < bw(v)$ **then**
$be(v), bw(v) := \{v, x\}, b(v)$
fi $\rangle$
rof $\bullet$
if $\neg core(v)$ **then**
$send(v)(dest(v)(up(v))(\text{REPORT}(bw(v), l))$
fi $\bullet$
if $core(v)$ **then** *ChangeRootPath*
else
$(receive(v)(dest(v)(up(v)))(\text{CHANGEROOT}(l)) \bullet$
$up(v) := be(v) \bullet$ *ChangeRootPath*) **or**
$(receive(v)(dest(v)(up(v)))(\text{NOCHANGE}(l)) \bullet$
NoChangeRoot)
fi $\bullet$
$core(v) := false \bullet$
(**if** $con\_req(v)(dest(v)(be(v))$ **then**
$send(v)(dest(v)(be(v))(\text{CONNECT}(l)) \bullet$
$con\_req := false \bullet$
$(receive(v)(dest(v)(be(v))(\text{NOCONNECT}(l))$ **or**
$(receive(v)(dest(v)(be(v))(\text{CONNECT}(l)) \bullet$
$core(v) := v < dest(v)(be(v))$))
fi $\| CH(v)) \bullet$
if $core(v)$ **then**
if $bw(v) = \infty$ **then** $fn(v) := term$
else $fn(v) := v$ **fi**
else
$receive(v)(dest(v)(up(v))(\text{INITIATE}(fn(v), l))$
fi $\bullet$
BroadCastName(v)
until $fn(v) = term$

Figure 1:

like atomic broadcast protocols [CASD] are studied in our framework.

References

[AO] K.R. Apt, E.-R. Olderog, *Verification of sequential and concurrent programs*, Springer, 1991.

[BHG] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bor] O. Boruvka, O jistém problému minimálním, *Práca Moravské Přírodovědecké Společnosti*, 3, 1926. (In Czech.)

[CG] C. Chou and E. Gafni, Understanding and Verifying Distributed Algorithms Using Stratified Decomposition, *Proc. of the 7th Annual Symp. on Principles of Distributed Computing*, 1988.

[CM] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[CASD] F. Critian, H. Aghili, R. Strong, D. Dolev, Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement, *Proceedings 15th International Symposium on Fault-Tolerant Computing*, 1985.

[Dijk] E. Dijkstra, Two Problems in Connection with Graphs, *Numer. Math.*, Vol. 1, pp. 269-271, 1959.

[EF] T. Elrad and N. Francez, Decomposition of distributed programs into communication closed layers, *Science of Computer Programming 2*, 1982.

[GHS] R.T. Gallager, P.A. Humblet and P.M. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM TOPLAS 5-1*, 1983.

[JPZ] W. Janssen, M. Poel and J. Zwiers, Action Systems and Action Refinement in the Development of Parallel Systems, an Algebraic Approach, *proceedings CONCUR '91*, Springer LNCS 527, 1991.

[JZ] W. Janssen and J. Zwiers, Protocol Design by Layered Decomposition, A compositional approach, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 571, 1992.

[JZa] W. Janssen and J. Zwiers, From Sequential Layers to Distributed Processes: Deriving a Distributed Minimum Weight Spanning Tree Algorithm, *Memoranda Informatica*, University of Twente, 1992.

[KP] S. Katz and D. Peled, Verification of Distributed Programs using Representative Interleaving Sequences, to appear in: *Distributed Computing*.

[Kru] J. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, in: *Proc. of the American Mathematical Society*, Vol. 7, pp. 48-50, 1956.

[Lam] L. Lamport, The Hoare Logic of concurrent programs, *Acta Informatica 14*, 1980.

[MP] Z. Manna, A. Pnueli, Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, *Science of Computer Programming 4*, 1984.

[OG] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informatica 6*, 1976.

[Pratt] V. Pratt, Modelling Concurrency with Partial orders, *International Journal of Parallel Programming 15*, 1986, pp. 33-71.

[Pri] R. Prim, Shortest connection networks and some generalizations, *Bell Syst. Tech. Journal*, Vol. 36, pp. 1389-1401, 1957.

[PZ] M. Poel and J. Zwiers, Layering Techniques for the Development of Parallel Systems, An Algebraic Approach, to appear in: *Proc. Computer Aided Verification*, 1992.

[Rey] J.C. Reynolds, *The Craft of Programming*, Prentice Hall, 1981.

[SR] F.A. Stomp and W.P. de Roever, Designing distributed algorithms by means of formal sequentially phased reasoning, *Proc. of the 3rd International Workshop on Distributed Algorithms*, Nice, LNCS 392, Eds. J.-C. Bermond and M. Raynal, 1989, pp. 242-253.

[Sto] F.A. Stomp, A derivation of a broadcasting protocol using sequentially phased reasoning, in: *Stepwize Refinement of Distributed Systems*, J.W. de Bakker et al. (Eds), Springer LNCS 430, 1989.

[Tar] R.E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Apllied Mathematics, 1983.

[WLL] J.L. Welch, L. Lamport, N. Lynch, A Lattice-Structured Proof Technique Applied to a Minimum Weight Spanning Tree Algorithm, *Proc. of the 7th Annual Symp. on Principles of Distributed Computing*, 1988.

[Zwi] J. Zwiers, Layering and Action Refinement for Timed Systems, in: *Proc. REX Workshop on Real Time: Theory in Practice*, Springer Lecture Notes in Computer Science, 1992.

Progress Measures and Stack Assertions for Fair Termination

Nils Klarlund*
IBM T.J. Watson Research Center
PO BOX 704
Yorktown Heights, New York

Abstract

Floyd's method based on well-orderings is the standard approach to proving termination of programs. Much attention has been devoted to generalizing this method to termination of programs that are subjected to fairness constraints. Earlier methods for fair termination tend to be somewhat indirect, relying on program transformations, which reduce the original problem to several termination problems.

In this paper we introduce the new concept of stack assertions, which directly—without transformations—quantify progress towards fair termination. Moreover, we show that by one simple program transformation of adding a history variable, usual assertional logic, without fixed-point operators, is sufficiently expressive to form a sound and relatively complete method when used with stack assertions. This result is obtained as part of a substantial simplification of earlier completeness proofs.

1 Introduction

Fairness is the assumption that an action that is enabled over and over will eventually be taken. Such assumptions are central to many distributed or concurrent systems. The fair termination problem—how to prove that a program terminates under assumption of fairness—is typical to much reasoning with fairness, and many methods for this problem have been suggested; see [AO83, AFK88, DH86, FK84, Fra86, GFMdRv85, LPS81, MP91, SdRG89]. Most of the methods build on Floyd's approach of using well-ordered sets as a measure of how close the program is to termination. Floyd's ideas allow one to annotate the unaltered program with assertions expressing closeness to termination, whereas many of the earlier methods for fair termination depend on changing the program. The modifications either consist of adding new program variables and unbounded nondeterminism, or involve recursively applied proof rules that transform the program. Unfortunately, these modifications tend to obscure how each step of the original program contributes to fair termination.

The goal of this paper is a lucid and practical approach for showing fair termination without repeated or drastic transformations. Our approach is based on the novel concept of *progress measure* introduced in [Kla90]; see also [Kla, KK91, Kla91, KS93]. A progress measure is a function on the states (or histories) of the *unaltered* program. The value of the function for a given state quantifies—in a certain mathematical sense—how close that state is to satisfying a property about infinite computations. The property is defined by a *specification*, which characterizes states (or histories), and by a *limit condition*, which when applied to the specification defines the allowed infinite computations.

The property, for example, could be that every infinite computation is unfair—this means that the program fairly terminates. In this case, the specification characterizes for each state which actions are enabled and which are taken; the limit condition expresses the fixed temporal meaning of unfairness: *some* action *is* enabled infinitely often while being taken only finitely often.

An essential property of a progress measure is that on every program transition, its value changes in a way ensuring that the computation converges according to the limit condition. This requirement can be formulated as *verification conditions*, which allow verification of *global* properties (on infinite computations) in terms of *local* reasoning (about states and transitions).

*This work was mainly carried out while the author was with the IBM T.J. Watson Research Center. The work has also been supported by an Alice & Richard Netter Scholarship of the Thanks to Scandinavia Foundation, Inc.; Forskerakademiet, Denmark; and Esprit Basic Research Action Grant No. 3011, Cedisys. Author's current address: Aarhus University, Department of Computer Science, Ny Munkegade, DK-8000 Aarhus, Denmark. E-mail: klarlund@daimi.aau.dk.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0229...$1.50

This correspondence is especially meaningful if the local reasoning can be done for an unmodified program, in which case it becomes clear what the exact contribution of each program step (by each process) is towards the global behavior. In this paper we provide a practical tool, called a *stack assertion*, that provides such an understanding for distributed or concurrent programs that fairly terminate. The stack assertions of a program define a mapping, called a *fair termination measure*, that describes how close each program state is to fair termination.

The contributions of this paper are both practical and theoretical. We demonstrate the usefulness of stack assertions by examples. For distributed or concurrent programs, our examples indicate a direct way of contributing "lack of progress towards termination" to "progress towards unfair execution" as expressed by a hierarchy of unfairness hypotheses. Stack assertions form the natural framework for expressing this hierarchy and summarize in a single data structure the information obtained by the program transformations of previous methods. Since the need for transformations has been eliminated, stack assertions can be added to existing assertional methods for concurrent and distributed programs.

There are two theoretical results of this paper. The first is a new completeness proof—substantially simpler than earlier proofs that involve transfinite induction or results from topology—that explains why a fair termination measure always exists for programs or distributed systems that fairly terminate.

The second result is that by adding a history variable to a program, the fair termination measure can be expressed by means of stack assertions in any reasonably expressive assertion language (i.e. a language that includes arithmetic).

In some earlier work on expressing assertions about fair termination [SdRG89, Mor90, MP91], predicate calculus is combined with fixed-points and ordinals. For an arbitrary program, this calculus allows to characterize precisely the states from which all infinite computations are unfair.

In the present paper we show that with the addition of a history variable, an assertion language containing only predicate calculus is sufficient for a proof of fair termination from the initial state. In our method, well-foundedness is expressed not by fixed-point logic in program assertions, but as an additional requirement that a relation, expressed by the program assertions, is well-founded (has no infinite descending chains.) Observe that adding history information (as for example is also done in methods for verification with nondeterministic automata [AL91, Sis91]) is a benign transformation in that it has to be done only once and basically does not change the transitional structure of the program; in particular, no additional nondeterminism is added.

2 Verification Methods for Fairness

A *fairness constraint* partitions infinite computations into fair and unfair ones. In this paper we shall concentrate on *strong fairness*, which is one of the most important fairness concepts. According to this criterion, a computation is *fair* if commands (or processes, statements, actions, events,...) that are enabled infinitely often are also executed infinitely often. (It is assumed that the number of different commands is finite.) Thus an *unfair* computation is one where some command is enabled infinitely often but only executed finitely often. A program P *fairly terminates* if every infinite computation of P is unfair. A *verification method* for fair termination is defined in terms of verification conditions expressed in the style of Hoare's logic. To be useful a method must be *sound*, i.e. any program for which the verification conditions can be satisfied must fairly terminate. The method is *complete* if the verification conditions can be satisfied for any program that fairly terminates.

Complete verification methods for strongly fair termination are considered in [GFMdRv85, LPS81, SdRG89]. These methods are based on *helpful directions*, which indicate program statements that are being unfairly executed. The approach of helpful directions has been successful at explaining many fairness concepts, such as those involving general state predicates [FK84] or an infinite number of commands [Mai89]. All these methods involve the recursive use of proof rules that are applied to transformed programs. Thus they tend to depend on particular syntactic properties of the underlying program language [Fra86, page 117] (although a way of circumventing these syntactic dependencies is indicated in [Fra86, section 2.4]).

The methods of *explicit schedulers* developed in [AO83, APS84, DH86] involve transforming programs by adding auxiliary variables that are nondeterministically assigned values determining fair computations. Because they involve rather drastic—even "cruel" [DH86]—program transformations, these methods also deal with fairness in a somewhat indirect manner. For an extensive treatment of fairness based on helpful directions and explicit schedulers, see the book [Fra86].

In [SdRG89] it was shown how predicate calculus augmented with fixed-points can be used to express assertions about fair termination. This calculus can express *inductively definable relations* [Mos74], which

are needed in their proof.[1] Usually, however, assertional reasoning is based on ordinary predicate calculus, which corresponds to *arithmetic relations* [Rog67]. Earlier, Apt and Plotkin, motivated by the relationship mentioned above between fairness and nondeterminism, gave a semantic model for countable nondeterminism [AP86]. In addition, they provided a relatively complete proof system for termination, also based on fixed-point logic.

Using a fragment of fixed-point calculus, Manna and Pnueli formulated elegant proof rules for assertional reasoning about properties expressed in temporal logic. For the problem of fair response (which generalizes fair termination), they exhibited a simple proof rule, which is recursively applied to transformed programs.

Morris [Mor90] also used fixed-point calculus in his formulation of a weakest precondition semantics for fair termination of tail-recursive programs.

The work presented here is also related to the theory of automata on infinite words. In fact, the condition of fair termination is but an instance of a *Rabin pairs condition*, see [KK91], which is a requirement in a special disjunctive normal form about the infinite occurrence of states. The proofs in the present paper could have been formulated for Rabin pairs conditions (thus yielding a method for *general fairness* [FK84]), but for simplicity of exposition we have used conditions pertaining to strong fairness.

The *Rabin progress measures* in [KK91] express progress towards satisfaction of a Rabin pairs conditions. Applied to fair termination, a Rabin progress measure maps program states into a special kind of colored trees. This gives a concise method for fair termination that does not depend on program transformations [KK91]. The method is not entirely practical, however, because there is no natural way to describe the mapping into the colored tree, which has to be described explicitly—obstacles that are overcome in this paper. For a more detailed comparison, see Section 5.

A concept similar to our stacks is used in [Saf92], where the problem is to determinize an automaton with a *Streett condition* (a special conjunction) or, equivalently, to express that a Rabin condition holds along all computations of a nondeterministic automaton by means of a deterministic automaton. For complementation of tree automata, the *last appearance record* of [GH82] serves a purpose different from that of our stacks, namely to keep enough information about the past to make finitely represented choices in a winning strategy for a game that is won by satisfying a conjunction (such as a Streett condition). On the other hand, stacks in the form of Rabin progress measures can be used to show that there is no need for such information when the game is won by a disjunction (Rabin condition) [Kla92].

Finally, our work is related to more general techniques for proving liveness properties. Harel showed that by transformations on trees representing programs one can do program verification for all finite levels of the Borel hierarchy [Har86]. Using an automata-theoretic approach, Vardi gave a verification method for very general properties, including the Borel hierarchy [Var87]. In Vardi's framework, progress is measured relative to a nondeterministic automaton that defines incorrect computations. In contrast, the *progress measures* of [Kla90, Kla] are functions that relate the program state or history to a finite computation of a correctness specification. With this approach nondeterminism must be eliminated, since it makes it difficult to relate program to specification by means of a function (cf. the work [AL91, KS93, Sis91] on relating automata defining safety properties). Instead more powerful limit conditions than those usually studied (e.g. Rabin or Streett conditions) are used to define the infinite computations as limits of finite ones.

3 Stack Assertions

In this section we review the method of Floyd and explain how assertions can define a measure of progress for termination. We argue informally how stack assertions can be used to guarantee fair termination. For simplicity we present our examples using the language of guarded commands, but our technique is syntax-independent and also applies to strong fairness expressed for other formalisms describing distributed or concurrent systems.

3.1 Floyd's Method

For programs occurring in practice it is usually straightforward to quantify progress towards termination. This is done in terms of well-founded sets as first advocated by Floyd [Flo67]. A *well-founded set* $(W, \succ)$ is a set W with a binary relation $\succ$ such that there is no infinite descending sequence $w_0 \succ w_1 \succ \cdots$. For an example of proving termination, take the program

$$P1 : *[\ x < y \ \rightarrow x := x + 1\ ]$$

[1] The inductively definable relations are the same as the Π^1_1 relations. Π^1_1 is the class of relations on the form $\forall \alpha : p$, where α is a second-order object (such as an infinite computation) and p is a first-order formula (such as the one expressing that a computation is unfair). The problem of fair termination is Π^1_1-complete as is the problem of termination (of programs with countable nondeterminism).

consisting of a loop with a single guarded command, which is executed as long as its guard, $x < y$, is enabled (true). The variables take on integer values. To argue that $P1$ terminates, we use the mapping $\mu^{\mathrm{T}} = \max\{y-x, 0\}$ from program states into the well-founded set of natural numbers $0 < 1 < 2 < \cdots$. Here and in the sequel, the letter "T" refers to the hypothesis that the programs terminates. The mapping μ^{T} can be called a *termination measure*, since its value decreases with each iteration. The existence of a termination measure μ^{T} guarantees that P terminates, because an infinite computation $p_0, p_1, \ldots$ would produce an infinite descending sequence $\mu^{\mathrm{T}}(p_0) > \mu^{\mathrm{T}}(p_1) > \cdots$, contradicting the well-foundedness of the natural numbers.

In practice, the termination measure μ^{T} is expressed by annotating the program with assertions. For $P1$ a single assertion suffices:

$$P1'\colon *[\ \left(\mathrm{T} : \max\{y-x, 0\}\right) \quad x < y \ \rightarrow x := x+1\ ]$$

Here $\left(\mathrm{T} : \max\{y-x, 0\}\right)$ is a simple *stack assertion*. It asserts that for the termination hypothesis, also called the T-*hypothesis*, the value of the termination measure μ^{T} is $\max\{y-x, 0\}$ whenever the loop is to be executed. Thus it could be called a *loop variant*[2] as opposed to a *loop invariant*. The latter expresses a relationship between variables that is preserved under iterations.

3.2 Fair Termination

With a trivial modification of $P1$, proving termination is suddenly more intricate. Consider the program

$$P2\colon *[\ \ell_a\colon x < y \ \rightarrow x := x+1 \ \square \quad \ell_b\colon x < y \ \rightarrow \mathtt{skip}\ ]$$

where the loop is executed as long as $x < y$ by execution of either of the guarded commands $x < y \rightarrow x := x+1$ and $x < y \rightarrow \mathtt{skip}$. The choice is made nondeterministically. This program will not terminate if the second command is always chosen from some point on. Under assumption of (strong) fairness, however, $P2$ always terminates, because in an infinite computation of $P2$, ℓ_a is only executed finitely often, but enabled infinitely often; thus the computation is unfair with respect to command ℓ_a.

[2]The terms *variant function* or *bound function* are also used [Gri81].

The preceding argument was formulated in terms of infinite computations. In contrast, assertional reasoning deals only with program states and single transitions. The key to assertional reasoning about fairness is:

> If there is no progress towards termination, this can be attributed to some statement being executed unfairly.

For example, when ℓ_b is executed, the T-hypothesis is not active since there is no progress towards termination. Instead, progress towards executing ℓ_a *unfairly* can be measured. To do this we reformulate the assertion by including the *unfairness hypothesis*, called the ℓ_a-*hypothesis*, that ℓ_a is executed unfairly. Syntactically, this is done by putting "ℓ_a" on top of the underlying T-hypothesis; thus we write

$$\left(\frac{\ell_a}{\mathrm{T} : \max\{y-x, 0\}}\right).$$

This stack assertion expresses a hierarchy in which the T-hypothesis is the underlying hypothesis and the rôle of the ℓ_a-hypothesis is to explain progress when the underlying hypothesis can not. The annotated program is now:

$$P2'\colon *[\ \left(\frac{\ell_a}{\mathrm{T} : \max\{y-x, 0\}}\right) \quad \ell_a\colon x < y \ \rightarrow x := x+1 \ \square \quad \ell_b\colon x < y \ \rightarrow \mathtt{skip}\ ]$$

Progress is made towards unfair execution in terms of the ℓ_a-hypothesis whenever ℓ_a is enabled but not executed. Note that for any iteration, either

- ($\mathrm{V_a}$) ℓ_a is enabled and not executed, and the underlying T-measure remains constant (when ℓ_b is executed); or
- ($\mathrm{V_T}$) measure μ^{T} decreases (when ℓ_a is executed).

We can now argue that $P2$ fairly terminates in terms of the local conditions ($\mathrm{V_a}$) and ($\mathrm{V_T}$) as follows. In an infinite computation, either from some point on ($\mathrm{V_a}$) always applies, or infinitely often ($\mathrm{V_T}$) applies. In the first case, ℓ_a is always enabled but never executed. Hence the computation is unfair with respect to ℓ_a. In the second case, it holds that each time ($\mathrm{V_T}$) applies, μ^{T} is decreased, and at the other times, when ($\mathrm{V_a}$) applies, μ^{T} is unchanged. This yields an infinite decreasing sequence of natural numbers, which is a contradiction.

Thus we have proved that for any infinite computation of $P2$, only the first case is possible, i.e. $P2$ fairly terminates. This argument will later be generalized to a

soundness result, which shows that a verification condition, similar to the local conditions above, always implies fair termination. For now, however, we motivate this general result by looking at more examples.

3.3 Progress Measures for Unfairness Hypotheses

A more complex situation arises if ℓ_a is sometimes not enabled when ℓ_b is executed. In this case the stack assertion $\left(\frac{\ell_a}{\mathrm{T} : \max\{y-x,0\}}\right)$ cannot be applied, because condition $(\mathrm{V_a})$ is then sometimes not fulfilled. If the program fairly terminates, however, we can use a progress measure μ^{ℓ_a} for the ℓ_a-hypothesis. For an example of this situation, take:

$$
\begin{array}{lll}
P3\colon *[\ \ell_a : x < y \wedge z \bmod 117 = 0 & \rightarrow x := x+1 & \square \\
\quad\ \ell_b : x < y & \rightarrow z := z-1 &]
\end{array}
$$

This program fairly terminates, because for any infinite computation, ℓ_a can only be executed finitely often and the value of z decreases by one each time ℓ_b is executed; thus ℓ_a is enabled infinitely often. We might annotate the program as follows:

$$
\begin{array}{lll}
P3'\colon *[\ \left(\frac{\ell_a : z \bmod 117}{\mathrm{T} : \max\{y-x,0\}}\right) & & \\
\quad\ \ell_a : x < y \wedge z \bmod 117 = 0 & \rightarrow x := x+1 & \square \\
\quad\ \ell_b : x < y & \rightarrow z := z-1 &]
\end{array}
$$

Here $a : z \bmod 117$ denotes that a progress measure $\mu^{\ell_a} = z \bmod 117$ is associated with the ℓ_a-hypothesis. The measure μ^{ℓ_a} is a measure of how close $P3$ is to a state in which ℓ_a is enabled. For each iteration of the loop, either

- $(\mathrm{V'_a})$ measure μ^{T} is unchanged, ℓ_a is not executed, and either the value of z was 0 (mod 117) before the execution of ℓ_b, in which case ℓ_a was enabled, or the value of z (mod 117) was between 1 and 116 and decreases by 1; or
- $(\mathrm{V'_T})$ measure μ^{T} decreases.

The local conditions $(\mathrm{V'_a})$ and $(\mathrm{V'_T})$ ensure that an infinite computation is unfair with respect to ℓ_a. Consider the corresponding infinite sequence of stacks. It must be the case that from some point on, $(\mathrm{V'_a})$ applies to each transition. Thus ℓ_a is only executed finitely often. If from some point on it is never enabled, then μ^{ℓ_a} decreases for each iteration thereafter, contradicting the well-foundedness of the natural numbers. Therefore, ℓ_a is enabled infinitely often, and we conclude that any infinite computation is unfair with respect to ℓ_a.

3.4 Unfairness of Several Commands

An even more challenging situation occurs when more than one command may be executed unfairly. If we add an empty guarded command to $P3$, we obtain:

$$
\begin{array}{lll}
P4\colon *[\ \ell_a : x < y \wedge z \bmod 117 = 0 & \rightarrow x := x+1 & \square \\
\quad\ \ell_b : x < y & \rightarrow z := z-1 & \square \\
\quad\ \ell_c : x < y & \rightarrow \texttt{skip} &]
\end{array}
$$

This program fairly terminates, because any infinite computation is unfair with respect to either ℓ_a or ℓ_b. To see this we use the loop variant from $P3'$ modified to explain the lack of progress when ℓ_c is chosen for execution. In that case there is progress neither towards termination nor towards executing ℓ_a unfairly. But there *is* always progress towards something when a program fairly terminates; in fact, when ℓ_c is executed, ℓ_b is a candidate for unfair execution because it is enabled but not executed. Thus we can put the ℓ_b-hypothesis—that ℓ_b is executed unfairly—on top of the T- and ℓ_a-hypotheses. The annotated program then becomes:

$$
\begin{array}{lll}
P4'\colon *[\ \left(\frac{\ell_b}{\frac{\ell_a : z \bmod 117}{\mathrm{T} : \max\{y-x,0\}}}\right) & & \\
\quad\ \ell_a : x < y \wedge z \bmod 117 = 0 & \rightarrow x := x+1 & \square \\
\quad\ \ell_b : x < y & \rightarrow z := z-1 & \square \\
\quad\ \ell_c : x < y & \rightarrow \texttt{skip} &]
\end{array}
$$

This annotation can be used as an argument why $P4$ fairly terminates in a way similar to the previous arguments.

Note that if earlier methods involving recursive proof rules had been used instead to show that $P4$ fairly terminates, it would have been necessary to reason about three different programs: the original and two syntactically derived programs.

4 Verification Conditions, Soundness, and Completeness

In the preceding section we developed a notation for reasoning about fairness. For each program we considered an arbitrary infinite computation and argued, using the associated stacks, that the computation was unfair. The rationale for using stack assertions, however, is to avoid reasoning about infinite computations. So to obtain an assertional verification method, we formulate verification conditions for the stacks expressed by the assertions. When these conditions are fulfilled for all transitions, we say that the stack assertions define a *fair termination measure*. We show

that if a program has a fair termination measure, then it fairly terminates. Thus the verification method of stack assertions is sound.

We also give a completeness result: when a program fairly terminates, it has a fair termination measure. Moreover, we show that under certain conditions (which are fulfilled if a history variable is added to the program) then the fair termination measure can be expressed as program assertions in a reasonably powerful assertion language.

4.1 The Verification Conditions

To formulate the verification conditions we need a few definitions. A *program* P defines a transition relation $\rightarrow$ on a countable set of program states; moreover, P defines a set of initial program states and a finite set of commands. A *command* (or an action, a process, an event,...) is designated by a label ℓ, and P defines for each program state whether ℓ is enabled or disabled. A *transition* $p \rightarrow p'$ describes the execution of exactly one command, which is enabled in p. A *path* from p_0 to p_n is a sequence of states $p_0, \ldots, p_n$ such that $p_i \rightarrow p_{i+1}$ for $i < n$; an infinite path $p_0, p_1, \ldots$ is defined in a similar way. A *computation* is a finite or infinite path starting in an initial state. A state p' is *reachable* from state p if there is a path from p to p'. We assume without loss of generality that any program state is reachable from an initial state. (In practice, conventional assertional methods can be used to describe the reachable program states, since finite sequences of program states can be encoded as numbers; see [MP91].)

A *progress hypothesis* or α-hypothesis is either an unfairness hypothesis, on the form ℓ or $\ell : w$ (with $\alpha = \ell$), or the T-hypothesis, on the form $T : w$ (with $\alpha = \ell$), where w is an element of a well-founded set $(W, \succ)$. A *stack assignment* is a mapping that maps each program state p to a list $\mu(p)$ of progress hypotheses such that the T-hypothesis is at level 0, i.e. at the bottom. (It can be assumed that all the hypotheses are different, i.e. there is at most one ℓ-hypothesis in $\mu(p)$ for each ℓ.) The stack assertions of a program define a stack assignment according to the semantics of the logical language of the assertions. (The exact correspondence is of no importance here.) For an hypothesis $\alpha : w$ in $\mu(p)$, where α is a label or "T," the value w is called the *α-measure* at p and is denoted $\mu^\alpha(p)$.

Note that the definitions above are not dependent on the particular syntax of guarded commands, but depend only on the notions of commands or actions being "enabled" and "executed." Thus our soundness and completeness results apply to strong fairness in all transition systems. For example, our method applies to nested commands ("all-level fairness," see [Fra86]).

The verification conditions are expressed in terms of active and non-invalidated hypotheses: essentially, an ℓ-hypothesis is *active* if progress towards unfair execution of ℓ is made, and it is *non-invalidated* if ℓ is not executed. The T-hypothesis is active if the program gets closer to termination; the T-hypothesis is always considered non-invalidated.

The verification conditions can now be stated somewhat informally:

(V_F) On any program transition,
- there is some active hypothesis;
- the active hypothesis and the ones below are non-invalidated;
- and the stack does not change below the active hypothesis.

The meaning of this is illustrated in Figure 1. Here the program transition is $p \rightarrow p'$. The active hypothesis α is at the same level in the stacks $\mu(p)$ and $\mu(p')$, and everything below (denoted by S in the figure) remains unchanged. Formally, the verification conditions (V_F) are:

(V_A) Some α-hypothesis is *active*, i.e. either
- α is a label ℓ and command ℓ is enabled (in state p or p'), or
- $w = \mu^\alpha(p)$ and $w' = \mu^\alpha(p')$ are defined with $w \succ w'$.

(V_{NonI}) Every hypothesis below and including the α-hypothesis is *non-invalidated* i.e. none of these hypotheses is the ℓ-hypothesis, where ℓ is the command executed in going from p to p'.

(V_{NoC}) The stack does not change below hypothesis α.

The contents of the stack above α may change in any way. When the stack assignment μ satisfies these conditions for all program transitions, we say that $(\mu, (W, \succ))$—or μ (when the well-founded relation $(W, \succ)$ is understood from the context)—is a *fair termination measure*.

4.2 Example

Here is an argument explaining why the stack assertion of $P4'$ satisfies (V_F). Consider an iteration not leading to termination. There are three cases depending on which command is executed:

ℓ_a: The T-hypothesis is active, because $\mu^T = \max\{y - x, 0\}$ decreases (since $x < y$ holds before ℓ_a is executed). There is nothing beneath the T-hypothesis to check.

ℓ_b: Below the ℓ_a-hypothesis, the stack remains unchanged and the T-hypothesis is not invalidated.

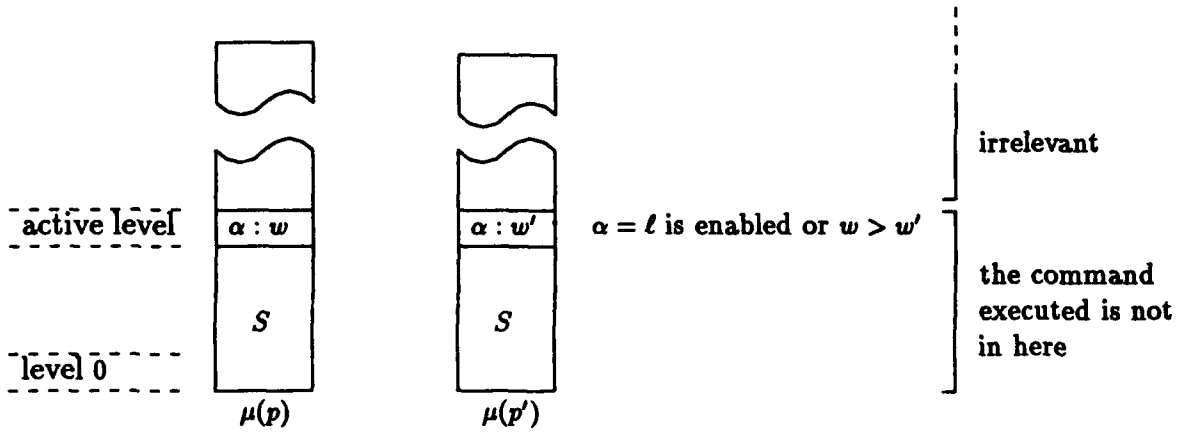


Figure 1: Verification condition.

The ℓ_a-hypothesis is active, because ℓ_a is not executed, and before execution of ℓ_b, either $z = 0 \pmod{117}$ holds, i.e. ℓ_a is enabled, or $z \neq 0 \pmod{117}$ holds, i.e. $\mu^{\ell_a} = z \bmod 117$ decreases.

ℓ_c: The stack is unchanged below the ℓ_b-hypothesis. The ℓ_b-hypothesis is active, because ℓ_b is enabled but not executed. The ℓ_a-hypothesis is non-invalidated, because ℓ_a is not executed.

4.3 Soundness

Theorem 1 (Soundness of Fair Termination Measures) If P has a fair termination measure, then P fairly terminates.

(See the Appendix for all proofs.)

4.4 Completeness

Theorem 2 (Completeness of Fair Termination Measures) If P fairly terminates, then P has a fair termination measure.

To prove Theorem 2, we first present a simple completeness proof, which applies to programs that are tree-like. A program is *tree-like* if it has a single initial state p^0 and if every state p', except p^0, has exactly one predecessor, i.e. there is exactly one p such that there is a transition $p \rightarrow p'$. Any program can be made tree-like by adding a history variable recording the past sequence of program states.

Theorem 3 If P fairly terminates and is tree-like, then P has a fair termination measure.

To prove Theorem 2 for an arbitrary program P, we apply Theorem 3 to the tree-like program P' that is obtained by adding a history variable to P. The value of the progress measure for a state p of P is then chosen as the least value of the progress measure of states in P' that correspond to p; here "least" means least with respect to a cross-product ordering on the progress measures of P'.

4.5 Relative Completeness

It is not hard to see that the completeness result in Theorem 3 can be sharpened to show that an effectively represented fair termination measure exists for an *effectively represented program* P (a program that has a recursive transition relation[3] and a recursive function that for all p' defines the state p (if it exists) such that $p \rightarrow p'$.) In fact, this measure can be obtained uniformly from P. To see this we define a *fair termination semi-measure* $(\mu, (W, \succ))$ to be a fair termination measure *except* that W need not be well-founded; thus μ is just required to satisfy the verification conditions.

Theorem 4 There is a recursive function h that given an index for a tree-like program P gives indices for a fair termination semi-measure $(\mu, (W, \succ))$, where both μ and $(W, \succ)$ are recursive. Moreover, $(\mu, (W, \succ))$ is a fair termination measure (i.e. $(W, \succ)$ is well-founded) iff P is fairly terminating.

This theorem gives an explicit reduction of the fair termination problem to a classical Π^1_1-complete problem of whether a recursive relation is well-founded. Moreover, it shows that if the assertional language includes usual predicate logic on numbers (and therefore all relations in the arithmetic hierarchy by a fundamental result of Gödel, see [Rog67]), then there exists a stack assertion

$$\left(\begin{array}{c} \alpha_N : w_N \\ \hline \cdots \\ \hline \alpha_1 : w_1 \\ \hline \mathrm{T} : w_T \end{array}\right),$$

[3] A *recursive relation* is also called a recursively computable relation, see [Rog67].

where the α's and w's are definable in the assertional logic, that satisfies the verification conditions.

Thus we obtain:

Corollary 1 (Relative Completeness of Stack Assertions) If the assertional language contains predicate calculus and if P fairly terminates, then P can be annotated with stack assertions in terms of the program history such that the verification conditions are satisfied.

5 Discussion

The results presented here are related to the method of helpful directions [Fra86, GFMdRv85, LPS81] and the Rabin measures of [KK91].

Formulated in our terminology, the method of helpful directions is used to identify one level of the fair termination measure at a time. For example, one first identifies subsets of program states corresponding to a constant μ^T measure. Then the program is transformed into several new programs, each corresponding to a subset. The states of each derived program are then further partitioned according to unfairness hypothesis (helpful directions) of the first level to yield more subsets, which are expressed as more derived programs.

Our approach is also related to the Rabin progress measures of [KK91, Kla90]. A Rabin progress measure is defined as a mapping from the program states into a colored tree. This mapping can be described in program assertions by specifying the progress values for each program state. The problem is that the colored tree has to be explicitly described (as it was done in an example given in [KK91]). In contrast, the stack assertions given in this paper are self-contained.

There are some technical differences that have been introduced to make stack assertions more useful for program annotation:

- Two stacks may contain the same progress values, but be colored differently. In a Rabin progress measure the coloring is a function of the progress values. Thus it is not possible to translate directly a fair termination measure into a Rabin progress measure.
- For a Rabin progress measure, satisfaction of an enabling condition is expressed in terms of the new state. For stack assertions, the satisfaction of the enabling condition is considered in terms of the old state and the new state.
- There may be several choices for an active hypothesis. For Rabin progress measures the active hypothesis is uniquely determined for each transition.

Acknowledgements

Thanks to Martín Abadi for very helpful comments. Dexter Kozen, Fred B. Schneider, and Mike Slifker also provided valuable comments on earlier versions of this paper.

References

[AFK88] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AO83] K.R. Apt and E.-R. Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3:65–100, 1983.

[AP86] K.R. Apt and G.D. Plotkin. Countable nondeterminism and random assignment. *JACM*, 33(4):724–767, 1986.

[APS84] K.R. Apt, A. Pnueli, and J. Stavi. Fair termination revisited with delay. *Theoretical Computer Science*, 33:65–84, 1984.

[DH86] I. Dayan and D. Harel. Fair termination with cruel schedulers. *Fundamenta Informatica*, 9:1–12, 1986.

[FK84] N. Francez and D. Kozen. Generalized fair termination. In *Proc. 11th POPL, Salt Lake City*. ACM, January 1984.

[Flo67] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science XIX*, pages 19–32. American Mathematical Society, 1967.

[Fra86] Nissim Francez. *Fairness*. Springer-Verlag, 1986.

[GFMdRv85] O. Grumberg, N. Francez, J.A. Makowsky, and W.P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66(1/2):83–102, 1985.

[GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proceedings 14th Symp. on Theory of Computing*. ACM, 1982.

[Gri81] David Gries. *The Science Of Programming*. Springer-Verlag, 1981.

[Har86] D. Harel. Effective transformations on infinite trees with applications to high undecidability, dominos, and fairness. *Journal of the ACM*, 33(1):224–248, 1986.

[KK91] N. Klarlund and D. Kozen. Rabin measures and their applications to fairness and automata theory. In *Proc. Sixth Symp. on Logic in Computer Science*. IEEE, 1991.

[Kla] N. Klarlund. Liminf progress measures. In *Proc. of Mathematical Foundations of Programming Semantics 1991*. To appear in LNCS.

[Kla90] Nils Klarlund. *Progress Measures and Finite Arguments for Infinite Computations*. PhD thesis, TR-1153, Cornell University, August 1990.

[Kla91] N. Klarlund. Progress measures for complementation of ω-automata with applications to temporal logic. In *Proc. Foundations of Computer Science*. IEEE, 1991.

[Kla92] N. Klarlund. Progress measures, immediate determinacy, and a subset construction for tree automata. In *Proc. Seventh Symp. on Logic in Computer Science*, 1992. To appear.

[KS93] N. Klarlund and F.B. Schneider. Proving nondeterministically specified safety properties using progress measures. To appear in *Information and Computation*, 1993.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: the ethics of concurrent termination. In *Proc. 8th ICALP*. LNCS 115, Springer-Verlag, 1981.

[Mai89] M.G. Main. Complete proof rules for strong fairness and strong extreme-fairness. Technical Report CU-CS-447-89, Department of Computer Science, University of Colorado, 1989.

[Mor90] J.M. Morris. Temporal predicate transformers and fair termination. *Acta Informatica*, 27:287–313, 1990.

[Mos74] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[MP91] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83:97–103, 1991.

[Rog67] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Book Company, 1967.

[Saf92] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proc. 24th Symposium on Theory of Computing*, 1992.

[SdRG89] F.A. Stomp, W.P. de Roever, and R.T. Gerth. The μ-calculus as an assertion-language for fairness arguments. *Information and Computation*, 82:278–322, 1989.

[Sis91] A.P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39(1):45–50, July 1991.

[Var87] M. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proc. Symp. on Logic in Computer Science*. IEEE, 1987.

Appendix: Proofs

Proof of Theorem 1

Assume that P has a fair termination measure $\mu(p)$ and that $p_0, p_1, \cdots$ is an infinite computation. We must prove that $p_0, p_1, \cdots$ is unfair. To see this we let κ_i be the level of the active hypothesis of the transition $p_i \rightarrow p_{i+1}$ and we define $\kappa = \liminf_{i\to\infty} \kappa_i$, i.e. κ is the least value of κ_i that occurs infinitely often. Then from some point on κ_i is always at least κ, i.e. there is a K such that for all $i \geq K$, $\kappa_i \geq \kappa$.

It is not hard to see that $\kappa > 0$; in fact, if κ was 0, then the values of the T-measure would form a sequence $\mu^{\mathrm{T}}(p_0) \succeq \mu^{\mathrm{T}}(p_1) \succeq \cdots$ (by $(\mathrm{V_A})$ and $(\mathrm{V_{NoC}})$),[4] where infinitely often the inequality is strict, namely each time $\kappa_i = 0$. This contradicts that $(W, \succ)$ is well-founded.

Thus $\kappa > 0$ and there is an ℓ such that for all $i \geq K$, the hypothesis at level κ is an ℓ-hypothesis (by $(\mathrm{V_{NoC}})$) and this hypothesis is non-invalidating (by $(\mathrm{V_{NonI}})$). It follows that ℓ is executed only finitely often. To see that the computation is unfair with respect to ℓ, we now only have to prove that ℓ is enabled infinitely often.

Assume the opposite is true. Thus for some $H \geq K$, it holds for all $i \geq H$ that ℓ is not enabled and

[4] $w \succeq w'$ means that $w = w'$ or $w \succ w'$.

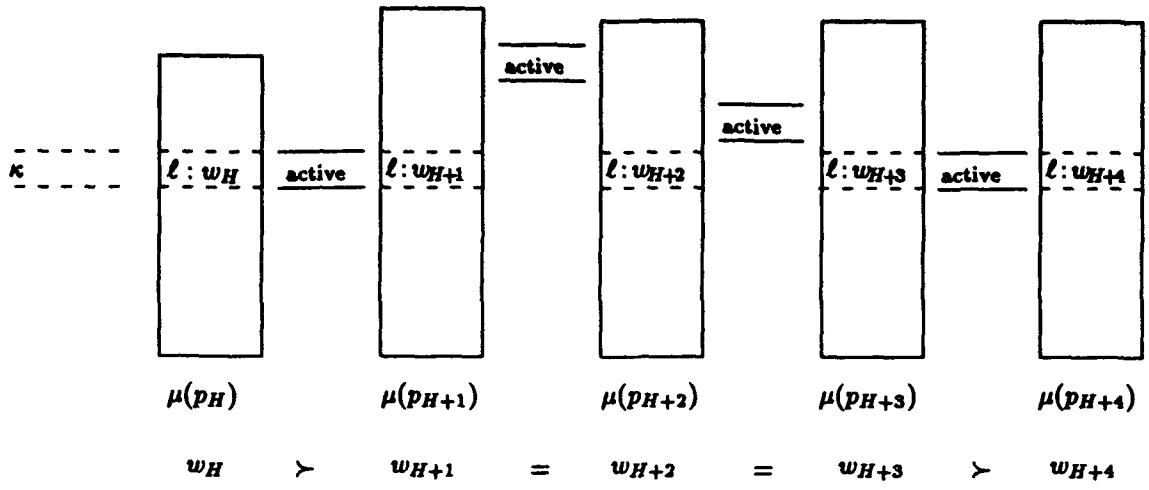


Figure 2: Soundness.

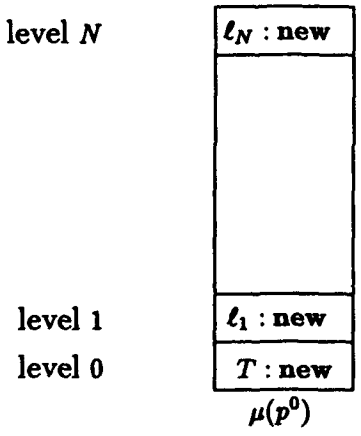


Figure 3: Initial stack.

the ℓ-hypothesis has the form $\ell : w_i$. As indicated in Figure 2, the values $w_i = \mu^\ell(p_i)$, $i \geq H$, give rise to an infinite descending sequence in W, because $w_H \succeq w_{H+1} \succeq \cdots$ (by (V_A) and (V_{NonI})) with strict inequalities whenever $\kappa_i = \kappa$. This contradicts that $(W, \succ)$ is well-founded.

Proof of Theorem 3

Assume that P fairly terminates and that there are N different commands. The proof is by a construction that defines the stack $\mu(p')$ in terms of the stack $\mu(p)$ when there is a transition $p \rightarrow p'$. Because the program is tree-like, this construction will define a unique value of μ for all p. The progress measures of the hypotheses take on values in a countable set W equipped with a relation $\succ$. Both W and $\succ$ are initially empty. The stack of p^0 is as illustrated in Figure 3. Here we created at levels 1 to N an hypothesis for each commands ℓ_i. The order of the hypotheses does not matter at this point. Each instance of new means that a new element is added to W. Hence creating the stack $\mu(p^0)$ results in there being $N+1$ elements in W, whereas $\succ$ remains empty.

When we create the stack $\mu(p)$ and use new to create a new element w at level κ, we define $\iota(w) = p$ and $\lambda(w) = \kappa$. Thus $\iota(w)$ denotes the program state where w is first used, and $\lambda(w)$ denotes the level where w is used.

Now assume that $\mu(p)$ has been defined and that there is a transition $p \rightarrow p'$ with ℓ denoting the command that is being executed. The idea behind the construction of $\mu(p')$ is to keep as much of $\mu(p)$ as possible. To state this more precisely we say that an ℓ'-hypothesis in $\mu(p)$ is *naturally active* if ℓ' is enabled in p or p' and the ℓ'-hypothesis is below the ℓ-hypothesis.

Case 1 If there is a naturally active hypothesis, let α be the naturally active hypothesis at the lowest level. The new stack becomes as illustrated in Figure 4. Here everything below α, indicated by S, is preserved. Also, the hypotheses above S are preserved, but their measures all change to new values.

Case 2 If there is no naturally active hypothesis, we let α be such that the α-hypothesis is the one just below the ℓ-hypothesis. Note that it may happen that $\alpha = \mathrm{T}$. The α-measure takes on a new value w', and we add $w \succ w'$ to the relation $\succ$ and say that α is *forced active*; in addition, the hypotheses above α are rotated one step downwards: Note that the ℓ is moved upwards (unless there is only one unfairness hypothesis in the stack) and that it is the only hypothesis moved upwards.

Whether $\mu(p')$ is constructed according to Case 1 or

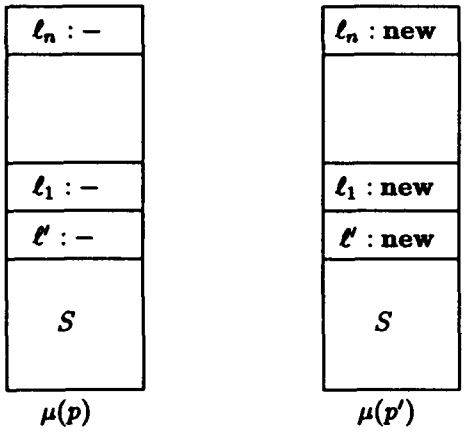


Figure 4: New stack in Case 1.

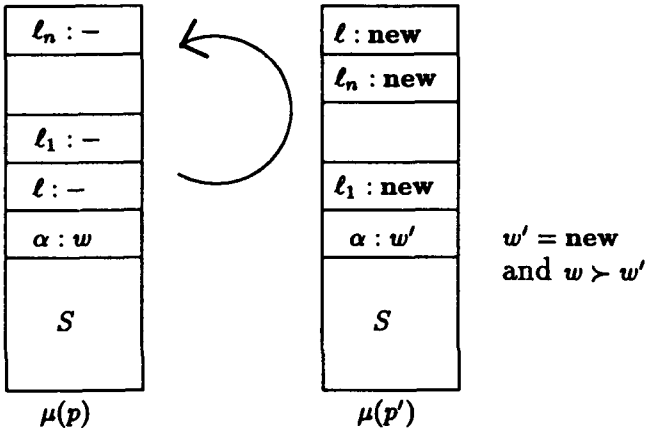


Figure 5: New stack in Case 2.

Case 2 above, the requirements (V_A), (V_{NonI}), and (V_{NoC}) can be seen to be satisfied for the transition $p \to p'$. Note also that when α is an active hypothesis, then there are no hypotheses below that can be active. Thus to finish the completeness proof we only need to show that $(W, \succ)$ is well-founded. We use the following properties:

Claim 1 If $p \to p'$, $\iota(w) \neq p'$, and $\mu^\alpha(p') = w$, then $\mu^\alpha(p) = w$ and the position of the α-hypothesis did not change on $p \to p'$. Moreover, if α is a label ℓ, then ℓ is not enabled and not executed on $p \to p'$. Also, the hypothesis just above the α-hypothesis in $\mu(p)$ does not change position and it is non-invalidated on $p \to p'$.

Proof By considering Case 1 and Case 2 above. □

Claim 2 Let w and w' be elements of W such that $w \succ w'$ and let $\kappa = \lambda(w)$. Let α by the hypothesis at level κ in $\mu(\iota(w))$.

(a) There is a path $\mathcal{P}^{w,w'} = p_0, \ldots p_n$ with $p_0 = \iota(w)$ and $p_n = \iota(w')$ such that the active level for $p_i \to p_{i+1}$ is greater than κ for $i < n-1$, and such that for $p_{n-1} \to p_n$ the hypothesis α is forced active.

(b) Moreover, no command of an hypothesis at or below κ is enabled along $\mathcal{P}^{w,w'}$.

Proof (a) By Claim 1, every p such that $\mu^\alpha(p) = w$ is reachable from $\iota(w)$ along a path where

- α is at level κ,
- if $\alpha = \ell \neq \mathrm{T}$, then ℓ is not executed and ℓ is not enabled, and
- $\mu^{\ell_\bullet}$ has the constant value w.

Since P is tree-like, there is a unique path $p_0, \ldots, p_{n-1}$ with $p_0 = \iota(w)$ such that there is a transition $p_{n-1} \to p_n = \iota(w')$, where $\mu(p_n)$ is constructed according to Case 2 and α is the hypothesis forced active.

(b) This follows from the choice of active hypothesis in Case 1 and Case 2. □

Now assume that there is an infinite descending sequence $w_0 \succ w_1 \succ \cdots$ in W. By (a) of the Claim, an infinite path $\mathcal{P}$ containing $\iota(w_0), \iota(w_1), \ldots$ can be put together from the paths $\mathcal{P}^{w_i,w_{i+1}}$. Along this path the active level is always at least $\kappa = \lambda(w_0) = \lambda(w_1) = \cdots$. Let α be the hypothesis at level κ. The commands that are executed infinitely often are above α, since (V_{NonI}) is satisfied. Also any command ℓ' that is executed only finitely often is eventually at level κ or below, because from the point where ℓ' is no longer executed, the ℓ'-hypothesis can only move downwards in the stack and will eventually settle at some level; this level is at most κ, because the hypotheses above κ are rotated infinitely often, namely each time α is forced active.

By the assumption that P fairly terminates, there is a command ℓ that is executed finitely often and enabled infinitely often. By the previous argument, the ℓ-hypothesis is at level κ or below. But the ℓ-hypothesis being infinitely often enabled then contradicts (b) of the Claim. Hence there are no infinite descending sequences in W, i.e. $(W, \succ)$ is well-founded. □

Proof of Theorem 2

The idea of the proof is similar to the use of the Sewing Lemma in [Kla92] for the immediate determinacy result of certain infinite games.

Assume that P fairly terminates. Also assume that there is a function $\mathcal{L}$ such that on any transition $p \rightarrow p'$, the value $\mathcal{L}(p')$ denotes the command executed in going from p to p' (the program state space and transition relation can always be extended to contain this information). By adding a history variable to P, we obtain a program $\overline{P}$, which also fairly terminates. A state of $\overline{P}$ is on the form $\sigma = \langle p_1, \ldots, p_n \rangle$ and the transitions of $\overline{P}$ are on the form $\langle p_1, \ldots, p_n \rangle \rightarrow \langle p_1, \ldots, p_{n+1} \rangle$, where $p_n \rightarrow p_{n+1}$ is a transition of P. The initial state of $\overline{P}$ is $\langle p^0 \rangle$, where p^0 is the initial state of P. For $\sigma = \langle p_1, \ldots, p_n \rangle$ define $p\sigma = p_n$. The set of states of $\overline{P}$ form a tree with root $\langle p^0 \rangle$. If $p \rightarrow p'$ and $p\sigma = p$, then the state $\sigma \cdot p'$ (which is list gotten by appending p' to the right end of σ) is a *child* of σ. A state σ is an *ancestor* of a state σ' if there are $\sigma_0, \ldots, \sigma_n$ such that σ_{i+1} is a child of σ_i for $i < n$ and $\sigma_0 = \sigma$ and $\sigma_n = \sigma'$. Define $\overline{\mathcal{L}}(\sigma) = \mathcal{L}(p\sigma)$. Let $\overline{\mu}$ designate the fairness measure given by the completeness proof of Theorem 3. The mapping $\overline{\mu}$ can be specified by a mapping α that to each σ associates a list $\overline{\alpha}(\sigma) = \langle \mathrm{T}, \ell_1, \ldots, \ell_N \rangle$ specifying the ordering of the hypotheses in the stack $\overline{\mu}(\sigma)$ and by a mapping $\overline{\theta} : \overline{P} \rightarrow W^N$ specifying for each σ a list $\mathbf{w} = \langle w_0, \ldots, w_N \rangle$ denoting the values of the progress measures at levels 0 to $N+1$. For a list $\mathbf{w}$, the i'th component is denoted $\mathbf{w}[i]$ and the sublist consisting of components from i to j is denoted $\mathbf{w}[i..j]$. We may assume that $(W, \succ)$ is totally ordered, i.e. is a well-ordering. We define an ordering, also denoted $\succ$, on W^{N+1} by $\mathbf{w} \succ \mathbf{w}'$ if for some i, $\mathbf{w}[i] > \mathbf{w}'[i]$, and for all $j < i$, $\mathbf{w}[j] = \mathbf{w}'[j]$. Then $\succ$ is a well-ordering. Now define $\theta(p) = \overline{\theta}(\sigma)$ and $\alpha(p) = \overline{\alpha}(\sigma)$, where σ is chosen such that $p\sigma = p$ and $\overline{\theta}(\sigma)$ is minimal with respect to $\succ$.

Claim 3 If $\overline{\theta}(\sigma)[0..n] = \overline{\theta}(\sigma')[0..n]$, then $\overline{\alpha}(\sigma)[0..n+1] = \overline{\alpha}(\sigma')[0..n+1]$.

Proof This follows from Claim 1. □

For $\mathbf{w}, \mathbf{w}' \in W^{N+1}$, define $|\mathbf{w}, \mathbf{w}'| = h$, where h is maximal such that for all $j \leq h$, $\mathbf{w}[j] = \mathbf{w}'[j]$.

Now consider a transition $p \rightarrow p'$ of P and let us prove that there is an α-hypothesis such that ($\mathrm{V_A}$), ($\mathrm{V_{NonI}}$), and ($\mathrm{V_{NoC}}$) are fulfilled. Let $\mathbf{w} = \theta(p)$ and $\mathbf{w}' = \theta(p')$. Then $\mathbf{w} = \overline{\theta}(\sigma)$ for some σ such that $p\sigma = p$. Let $\mathbf{w}'' = \overline{\theta}(\sigma \cdot p')$. By definition of $\theta(p')$, $\mathbf{w}'' \succeq \mathbf{w}'$. Also, let σ' be such that $\overline{\theta}(\sigma' \cdot p') = \theta(p')$.

There are two cases:

Case $\mathbf{w} \succ \mathbf{w}'$: Let $h = |\mathbf{w}, \mathbf{w}'|$. By Claim 3, $\alpha(p)[0..h+1] = \alpha(p')[0..h+1]$. Thus $\alpha(p)[h+1]$ is active and the stack below is unchanged, whence ($\mathrm{V_A}$) and ($\mathrm{V_{Noc}}$) are satisfied.

By definition of h, $\overline{\theta}(\sigma' \cdot p')[0..h] = \overline{\theta}(\sigma)[0..h]$. Thus the values in $\overline{\theta}(\sigma' p')[0..h]$ are created in an ancestor of $\sigma' \cdot p'$ and therefore $\overline{\theta}(\sigma')[0..h] = \overline{\theta}(\sigma' p')[0..h]$ by Claim 1. Also by Claim 1, it follows that $\mathcal{L}(p') = \overline{\mathcal{L}}(\sigma' \cdot p')$—the command executed on $p \rightarrow p'$—is not among $\overline{\alpha}(\sigma' \cdot p')[0..h+1] = \alpha(p')[0..h+1]$, whence ($\mathrm{V_{NonI}}$) is satisfied.

Case $\mathbf{w} \prec \mathbf{w}'$: We have $\mathbf{w}'' \succeq \mathbf{w}' \succeq \mathbf{w}$. Let $h = |w, w'|$. By construction of $\overline{\mu}$, the hypothesis at level $h+1$ is naturally active for the transition $\sigma \rightarrow \sigma \cdot p'$ of $\overline{P}$. But since $\mathbf{w}'' \succeq \mathbf{w}' \succeq \mathbf{w}$, it can be seen that $\mathbf{w}''[0..h] = \mathbf{w}'[0..h] = \mathbf{w}[0..h]$. It follows that $\alpha(p)[0..h+1] = \alpha(p')[0..h+1]$ and that the $\alpha(p)[h+1]$-hypothesis is naturally active for $p \rightarrow p'$ of P, whence ($\mathrm{V_A}$) and ($\mathrm{V_{NoC}}$) are satisfied. It can be seen in the same manner as in the previous case that ($\mathrm{V_{NonI}}$) is also satisfied □

Proof of Theorem 4

Given an effectively represented program P and a program state p, it is possible to calculate the sequence of program states, starting at the initial state, that leads to p. Thus the tree can be effectively traversed (even if it is infinitely branching). This traversal ensures that each time "new" is invoked, a unique progress value is returned. For example, we can represent W using the natural numbers; successive invocations of "new" then gives progress values '0,' '1,'...Note that the relation $\succ$ calculated on W is not the usual ordering on the natural numbers. Given a state p, the tree is traversed until p is encountered. At any program state, the value of the stack is calculated according to the the procedure given in the proof of Theorem 3. It follows that the value of the fair semi-measure at p can be recursively calculated. Similarly the relation $i \succ j$ can be seen to be recursive. By standard techniques of computability theory, the above procedure can be expressed formally as a recursive function h satisfying the properties in the statement of the Theorem. □

A Tradeoff Between Safety and Liveness for Randomized Coordinated Attack Protocols

George Varghese* Nancy A. Lynch †

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We study *randomized, synchronous* protocols for coordinated attack. Such protocols trade off the number of rounds (N), the worst case probability of disagreement (U), and the probability that all generals attack ($\mathcal{L}$). We prove a nearly tight bound on the tradeoff between $\mathcal{L}$ and U ($\mathcal{L}/U \leq N$) for a *strong* adversary that destroys any subset of messages. Our techniques may be useful for other problems that allow a non-zero probability of disagreement.

1 Introduction

Suppose two computers are trying to perform a database transaction over an unreliable telephone line. If the line goes dead at some crucial point, standard database protocols mark the transaction status as "uncertain" and wait until communication is restored to update its status. The protocol will ensure that the two computers *eventually* agree if communication is eventually restored.

On the other hand, suppose that the transaction has a real time constraint (e.g., a decision to commit or reject the transaction must be reached in 10 minutes) and the cost of disagreement is high. Then standard commit protocols do not work. If communication can fail for up to ten minutes it is always possible for the two computers to disagree. Is there a protocol that prevents disagreement in all cases?

The answer is no. The question was first formalized in [G] as the coordinated attack problem. In this problem, there are two generals who communicate only using unreliable messengers. The generals are initially passive; however, at any instant either general may get an input signal that instructs him to try to attack a distant fort. The generals have a common clock. The problem is to synchronize attack attempts subject to the conditions:

- **Validity**: If no input signal arrives, neither general attacks.[1]
- **Agreement**: Either both generals attack or they both do not attack.
- **Nontriviality**: There is at least one execution of the protocol in which both generals attack.

It is shown in ([G], [HM]) that there is no *deterministic* algorithm that meets all three conditions. In this paper, we consider a generalization to an arbitrary number of generals connected by a graph of unreliable links. Clearly the impossibility result applies here as well.

Coordinated attack (CA) looks suspiciously like Byzantine agreement (BA) [LPS]. The major differences are: first, in BA, *generals* exhibit arbitrary failures while in CA only *links* fail by destroying messages; second, in BA only *some fraction* of the generals are assumed to be faulty while in CA *all* links can be faulty. Thus there does not appear to be any way to reduce CA to BA or vice versa.

There is a well-known history of *randomization* providing a cure for a *deterministic* impossibility result

*Supported by DEC Graduate Education Program.
†Supported by NSF Grant 8915206-CR, DARPA Grant N0014-89-J1988, and ONR Grant N0014-91-J1046.

PoDC '92-8/92/B.C.
• 1992 ACM 0-89791-496-1/92/0008/0241...$1.50

[1] Another validity condition that is often used is that if no messages are delivered, then no general attacks. We prefer our definition because it focuses on input-output behavior. However, our results can be modified to fit the other validity condition.

(e.g. [RL], [B]). Thus we we turn to *randomized CA*. We hope to trade a small probability of disagreement when links fail for a high probability of agreement (on a positive outcome) when links do not fail.

We modify the correctness conditions for deterministic CA to fit randomized CA. We retain the validity condition. We modify the agreement condition by requiring that the worst case probability of disagreement (denoted by U) be smaller than ϵ, a parameter. We replace the nontriviality condition by a measure $\mathcal{L}(R)$ (for liveness) that measures the probability all generals attack after an input signal, given that messages are delivered according to a given pattern R.[2] We measure the goodness of a CA protocol by seeing how high $\mathcal{L}(R)$ can be for a given R and ϵ.

Coordinated attack captures the fundamental difficulty of real-time synchronization over unreliable message channels. This paper investigates whether randomization can help coordinated attack. Our answer is basically no for nontrivial adversaries, and a qualified yes for much weaker adversaries. Our paper concentrates on a *strong adversary* that can deliver messages according to any possible pattern R but has no access to message bits.[3]

The rest of this paper is organized as follows. Section 2 contains our model, Section 3 describes a simple but inefficient protocol, and Section 4 introduces some useful concepts. Section 5 contains a basic lower bound, Section 6 describes an optimal protocol against a strong adversary, and Section 7 contains a second, more refined, lower bound. Section 8 contains our conclusions and the appendix contains a proof of the second lower bound.

2 Model

The generals are represented by processes i that are at the vertices of a undirected graph $G(E, V)$ with $V = \{1, \ldots, m\}$, $m \geq 2$. We consider synchronous protocols that work in $N + 2$ rounds, numbered $-1, 0, \ldots, N$, $N \geq 1$. We model the input as a message sent at the end of a fictitious Round -1 and arriving at the end of Round 0 from a fictitious "environment" node v_0. We assume $v_0 \notin V$. Informally, if a process i receives a message in Round 0 from v_0, it has received a signal to try to attack. Each process i also receives a sequence of J random bits called α_i. J is an upper bound on the total number of random bits used by any general.

A *protocol* F consists of a number of *local protocols* F_i. Each $F_i, i \in V$, is a state machine executed by process i. F_i has two possible start states s_i^0 and s_i^1, a state transition function δ_i, and a message generation function σ_i. Let S_i^r be the set of messages *received* by i from its neighbors in round r. Let q_i^r be the state of i at the end of round r. Then $q_i^r = \delta_i(q_i^{r-1}, r, S_i^r, \alpha_i)$. We assume without loss of generality that processes send messages to each neighbor in rounds $1 \ldots N$ since we can always simulate algorithms in which this is not true by sending null messages that are ignored by the receiver. Let m_{ij}^r be the message *sent* by i to neighbor j in round r. Then $m_{ij}^r = \sigma_i(q_i^{r-1}, j)$. At the end of N rounds, i outputs a bit O_i based on q_i^N. $O_i = 1$ iff i decides to attack.

An execution of F is described in terms of a vector of local executions. A *local execution* E_i consists of q_i^0, (q_i^r, S_i^r, m_{ij}^r) for $1 \leq r \leq N$, and O_i. To generate an execution of F we need to define a run that represents the inputs as well as which messages get through in rounds $1 \ldots N$ of the protocol. Formally, a *run* $R = I(R) \cup M(R)$. $I(R)$, the *input* for run R, is an arbitrary subset of $\{(v_0, i, 0) : i \in V\}$. $M(R)$, the *messages delivered* in run R, is an arbitrary subset of $\{(i, j, r) : (i, j) \in E, 1 \leq r \leq N\}$. For example, in the run $\{(v_0, 3, 0), (1, 2, 6), (3, 2, 7)\}$ only F_3 receives a signal to attack. Also only the message sent in Round 6 from F_1 to F_2 and any message sent in Round 7 from F_3 to F_2 are delivered: all other sent messages are lost.

We will use the notation (A_i) to denote a vector A consisting of a component A_i for each $i \in V$. An execution for a fixed F is uniquely specified by random input $\alpha = (\alpha_i)$, and a run R. We define $Ex(R, \alpha) = (E_i)$ as the *execution generated* by R and α for a fixed protocol F. Each E_i is a local execution such that:

- If $(v_0, i, 0) \notin R$ then $q_i^0 = s_i^0$. If $(v_0, i, 0) \in R$ then $q_i^0 = s_i^1$ (i.e., the initial state of the local execution encodes the input).
- For all $r, 1 \leq r \leq N$: $m_{ij}^r = \sigma_i(q_i^{r-1}, j)$.
- For all $r, 1 \leq r \leq N$: $m_{ji}^r \in S_i^r$ iff $(j, i, r) \in R$.
- For all $r, 1 \leq r \leq N$: $q_i^r = \delta_i(q_i^{r-1}, r, S_i^r, \alpha_i)$.

The *output* of execution E is the vector $(O_i(q_i^N))$. We say two executions E and $\tilde{E}$ are *identical* to j if $E_j = \tilde{E}_j$.

[2] It may seem strange that unsafety is measured as the worst case across all runs while liveness is measured separately for each run. However, the situation is similar to Data Link protocols in which the prefix property (safety) is always preserved but liveness is guaranteed only if the channel is delivering messages.

[3] Since our lower bounds are pessimistic, there is no point in considering a stronger adversary that can read message bits. Also, some form of encryption could be used to make this assumption reasonable.

We consider sets of executions of a particular protocol. If X and Y are sets of executions, then XY denotes $X \cap Y$, and $X + Y$ denotes $X \cup Y$. D_i denotes the set of executions in which $O_i(q_i^N) = 1$, and $\overline{D_i}$ the set of executions in which $O_i(q_i^N) = 0$. Similarly, $(D_i|R)$ denotes the set of executions that have run R and in which $O_i(q_i^N) = 1$.

TA (*total attack*) denotes the set of executions $D_1 D_2 ... D_m$. NA (*no attack*) denotes the set of executions $\overline{D_1}\ \overline{D_2} \ldots \overline{D_m}$. PA (*partial attack*) denotes the complement of $NA \cup TA$. Thus, TA is the set of executions in which all processes agree on an output of 1, NA is the set of executions in which all processes agree on an output of 0, and PA is the set of executions in which some pair of processes disagree.

Each α_i is drawn from $\{0,1\}^J$ using the uniform probability distribution. This probability distribution on inputs α induces a probability distribution on executions for each possible run R, in the natural way. For each set X of executions and each run R, we use the notation $Pr[X|R]$ to denote the probability of event X according to this distribution of executions.

Now consider two runs $R = \{(i,j,1)\}$ and $\tilde{R} = \emptyset$. The only difference in the runs is that i sends a message that is delivered in R. Thus, given the same random input, i will decide the same regardless of whether an execution follows run R or run $\tilde{R}$. This leads to a key notion of indistinguishable runs. We say that two runs R and $\tilde{R}$ are *indistinguishable* to i if for all α, $Ex(R,\alpha)$ and $Ex(\tilde{R},\alpha)$ are identical to i. We use $R \stackrel{i}{\equiv} \tilde{R}$ to denote that R and $\tilde{R}$ are indistinguishable to i. A natural consequence is:

Lemma 2.1 If $R \stackrel{i}{\equiv} \tilde{R}$ then $Pr[D_i|R] = Pr[D_i|\tilde{R}]$.

An *adversary* $\mathcal{A}$ is a set of runs. We will only deal in this paper with a *strong adversary*, $\mathcal{A}_s$, where $\mathcal{A}_s$ is the set of all possible runs.

Next, we describe the correctness conditions and the liveness measure. Validity requires that no process attacks if there is no input. Agreement requires that the worst-case probability of partial attack be no more than ϵ, a parameter. Finally the liveness measure for a run R is the probability of total attack on run R.

- **Validity** : A protocol satisfies validity if for all vectors α, for all R such that $I(R) = \emptyset$, and for all i: $O_i = 0$ in $Ex(R,\alpha)$.
- **Agreement**: We define $U_{\mathcal{A}}(F)$, the *unsafety* of protocol F against adversary $\mathcal{A}$, as: $U_{\mathcal{A}}(F) = Max_{R \in \mathcal{A}} Pr[PA|R]$. Then F satisfies agreement with parameter ϵ if $U_{\mathcal{A}}(F) \leq \epsilon$.
- **Liveness**: We define *liveness* $\mathcal{L}(F,R)$ of protocol F on run R by: $\mathcal{L}(F,R) = Pr[TA|R]$.

Our goal is to find an "optimal" algorithm F that meets the validity and agreement conditions, and such that $\mathcal{L}(F,R)$ is as large as possible for any run R. We end this section with two elementary lemmas on which our lower bounds are based. The first states that the unsafety is at least as large as the difference in attack probabilities of any two processes. The second states that the liveness is no more than the attack probability of any process. The two inequalities given below do not seem very tight, and so it is perhaps surprising that the lower bounds based on these inequalities are as tight as they are.

Lemma 2.2 For all $i,j \in V$, $Pr[D_i|R] - Pr[D_j|R] \leq U_s(F)$.

Lemma 2.3 For all $i \in V$, $\mathcal{L}(F,R) \leq Pr[D_i|R]$.

3 Example Protocol

We informally describe a simple protocol A for two processes 1 and 2 against a strong adversary. The limitations of this protocol will motivate both the lower bound in Section 5 and the optimal protocol of Section 6.

In order to conform to the model, we require that each process must send some message (at least a null message) in every round. For convenience, let us call a non-null message (i.e., a message that carries information) a packet. We assume implicitly that on every round a process sends either a packet or a null message.

Initially, at the start of round 0, process 1 chooses a random integer *rfire* that is uniformly distributed between 2 and N. Process 1 includes the value of *rfire* in any packet it sends. If process 2 receives any packet from process 1, process 2 will store the value of *rfire*.

In rounds 1 through N, the two processes send packets to each other in alternate rounds. Process 2 is allowed to send packets in odd rounds starting from round 1, while process 1 is allowed to send packets in even rounds. The protocol begins with process 2 sending a packet in round 1. However, in all later rounds, a process sends a packet in a round only if it has received a packet in the previous round, and it is allowed to send a packet in the round. Thus if the adversary destroys a packet sent in round r, all packet sending stops in rounds greater than r.

The main idea is that if all packets sent strictly before round number *rfire*, have been delivered, then

the process that received the last packet (say i) will decide to attack. If the next packet sent by process i is delivered then the other process (say j) will also decide to attack. On the other hand, if any packet sent before round *rfire* is destroyed, then both processes stop sending packets and do not attack. Since the adversary that controls message delivery does not know the value of *rfire*, the adversary has only a chance of approximately $1/N$ of causing partial attack. This is because the adversary can cause partial attack *only if the first packet destroyed in the run is the packet sent in round rfire*. Thus $U_s(A) \approx 1/N$.

In addition, process 2 includes a bit that encodes its input in the packets it sends. Suppose at the end of Round 1, process 1 has not received a signal to attack and has not received a packet from process 2 saying that process 2 has received a signal to attack. Then process 1 does not send a packet in Round 2, and the protocol stops. Thus protocol A satisfies validity. Finally, let R_g be a "good" run in which all messages are delivered and the input is valid. Then on run R_g, both processes will always decide to attack. Hence $\mathcal{L}(A, R_g)$, the liveness of A on run R_g, is 1. However, this simple protocol raises two questions:

- $U_s(A) \approx 1/N$ and $\mathcal{L}(A, R_g) = 1$. Can we decrease $U_s(A)$ further while keeping $\mathcal{L}(A, R_g)$ unchanged? In other words, can we find a protocol a) whose probability of making a mistake is better than $1/N$, and b) whose probability of attacking on a good run is 1. It might seem that this can be done by running A several times. However, the answer is no, as we show in Section 5.
- Consider a run R in which the input is valid and all messages are delivered except the message sent by process 1 in Round 2. It is easy to see that $\mathcal{L}(A, R) = 0$. Intuitively, this is not satisfactory because in run R, all but one message is delivered, and yet the probability of attacking on run R is 0. Can we design a protocol whose liveness grows in some fashion with the number of messages delivered in a run? We will describe an "optimal" protocol S in Section 6.

4 Information Flow, Clipping, and Information Level

In this section, we describe three concepts that underlie both the lower bounds of Section 5 and the protocol in Section 6. We begin with a definition that captures the usual idea of information flow or possible causality [L] between process-round pairs in a run.

Consider any $i, k \in V \cup \{v_0\}$ and any $r, s \in \{-1, 0, \ldots, N\}$. We say that (i, r) directly flows to (k, s) in run R iff $s = r + 1$ and either $i = k$ or $(i, k, s) \in R$. We define the *flows to* relation between process-round pairs as the reflexive transitive closure of the directly flows-to relation. Thus:

Lemma 4.1 If (i, r) flows to (j, s) and (j, s) flows to (k, t) in run R, then (i, r) flows to (k, t) in run R.

We introduce a measure of the "knowledge" [HM] a process has in a run. We first define information "height" and use it to define the more useful idea of information "level". Intuitively, a process reaches height 1 when it hears the input. A process reaches height $h > 1$ when it has heard that *all* other processes have reached height $h - 1$. More formally, we say that j can reach height h by round r in run R iff h is a nonnegative integer subject to the following conditions:

- If $h = 0$, there are no conditions.
- If $h = 1$, $(v_0, -1)$ flows to (j, r) in R.
- If $h > 1$, then for all $i \neq j \in V$, there is some r_i such that (i, r_i) flows to (j, r) in R and i can reach height $h - 1$ by round r_i in R.

Next, we define $L_j^r(R)$, the *level* j reaches by round r of run R, to be the *maximum* height j can reach by round r. We use $L_j(R)$ to denote[4] $L_j^N(R)$ and $L(R)$ to denote $Min_{j \in V}(L_j(R))$.

Finally, we introduce a construction to "clip" a run with respect to a process i such that the constructed run preserves all information flow to i. This construction is the key to the lower bound proof. We define $Clip_i(R) = \{(j, k, r) \in R : (k, r) \text{ flows to } (i, N)\}$ in run R. It is not hard to see that clipping with respect to i preserves any information that i can gather in the run. Hence we have:

Lemma 4.2 Let $Clip_i(R) = \tilde{R}$. Then $L_i(R) = L_i(\tilde{R})$ and $R \stackrel{i}{\equiv} \tilde{R}$.

5 Lower Bound for Strong Adversary

The first lemma captures the intuitive idea that a change in level can only come about by receiving a message.

[4] Recall that N is the maximum round number

Lemma 5.1 For any run R and any $k \in V$, if $L_k(R) = l > 0$ then there must be some tuple $(j, k, r) \in R$ such that $L_k^r(R) = l$.

Proof: From the definition of level, we see that if there is no j, s such that $(j, k, s) \in R$ then $L_k^{s-1}(R) = L_k^s(R)$. Thus if $L_k^N(R) = l$ we can work backwards from round number N until we find the r required for the lemma. If we fail then there is no $(*, k, *)$ tuple in R, which would imply that $l = 0$, a contradiction. Thus we cannot fail. □

The next lemma describes the key property of clipped runs and information levels that we use to prove our lower bound. It says that if i reaches information level l at the end of run R then at the end of $Clip_i(R)$ there must be some process k whose information level is no more than $l-1$. In essence, this is why i cannot go to a higher information level than l by the end of R.

Lemma 5.2 Consider a run R such that $L_i(R) = l > 0$ and $Clip_i(R) = \tilde{R}$. Then there is some $k \in V$ such that $L_k(\tilde{R}) \leq l - 1$.

Proof: By contradiction. Thus for all $k \in V$, we assume that $L_k(\tilde{R}) \geq l$.

Consider any $k \neq i$. By Lemma 5.1 and the fact that $l > 0$, there must be some tuple $(j, k, r) \in \tilde{R}$ such that $L_k^r(\tilde{R}) \geq l$. Since $(j, k, r) \in \tilde{R}$ then (by definition of clipping), (k, r) flows to (i, N) in R. Hence, we can show that (k, r) flows to (i, N) in $\tilde{R}$. We also know that $L_k^r(\tilde{R}) \geq l$. Since this is true for all $k \neq i$ we must have (see the definition of level) $L_i(\tilde{R}) \geq l + 1$. But by Lemma 4.2, this implies that $L_i(R) \geq l + 1$, a contradiction. □

Lemma 5.3 For all protocols F, all runs R, and any process index $i \in V$, $Pr[D_i|R] \leq U_s(F)L_i(R)$.

Proof: By induction on l in the following inductive hypothesis.

Inductive hypothesis: For all i and all runs R with $L_i(R) = l$, $Pr[D_i|R] \leq U_s(F)l$.

Base case, $l = 0$: Thus $L_i(R) = 0$. Let $\tilde{R} = Clip_i(R)$. We first claim that $I(\tilde{R}) = \{\}$. Suppose not for contradiction. Then there is some j such that $(v_0, j, 0) \in \tilde{R}$; hence, since $\tilde{R} \subseteq R$, $(v_0, j, 0) \in R$. Also by the definition of clipping, $(j, 0)$ flows to (i, N) in R. But in that case, $L_i(R) \geq 1$, a contradiction. Thus we must have $I(\tilde{R}) = \{\}$. Also by Lemma 4.2, $R \overset{i}{\equiv} \tilde{R}$. Hence $Pr[D_i|R] = Pr[D_i|\tilde{R}] = 0$, by Lemma 2.1 and the validity requirement. Thus $Pr[D_i|R] = U_s(F)L_i(R)$.

Inductive Step, $l > 0$: Consider any l and R such that $L_i(R) = l$. Let $\tilde{R} = Clip_i(R)$. By Lemma 5.2, there exists some k such that $L_k(\tilde{R}) \leq L_i(R) - 1$. Hence, by the inductive hypothesis, $Pr[D_k|\tilde{R}] \leq U_s(F)(l-1)$. But by our bound on unsafety, Lemma 2.2, $Pr[D_i|\tilde{R}] - Pr[D_k|\tilde{R}] \leq U_s(F)$. Hence $Pr[D_i|\tilde{R}] \leq U_s(F)l$. But by the fact that R and $\tilde{R}$ are indistinguishable to i and by Lemma 2.1, it follows that $Pr[D_i|R] \leq U_s(F)l$. □

Theorem 5.4 For any F, $\mathcal{L}(F, R) \leq U_s(F)L(R) \leq \epsilon L(R)$.

From Lemma 5.3, for any $i \in V$, $Pr[D_i|R] \leq U_s(F)L_i(R)$. Thus from Lemma 2.3, $\mathcal{L}(F, R) \leq U_s(F)L_i(R)$ for any $i \in V$. Thus from the definition of $L(R)$, $\mathcal{L}(F, R) \leq U_s(F)L(R)$. The theorem now follows from the agreement condition. □

6 Optimal Protocol Against a Strong Adversary

In Protocol S which we describe below, we will arbitrarily designate process 1 to choose a random number *rfire*. In order to attack, we will require that any other process i hear the value of *rfire* from process 1 in addition to hearing the input. This motivates a second measure on a run R that we call the ***modified level*** measure. It is defined in a parallel fashion to the original level measure by first defining a modified height or m-height. Formally, we say that process j can ***reach m-height h*** by round r in run R iff h is a nonnegative integer subject to the following conditions:

- If $h = 0$, there are no conditions.
- If $h = 1$, $(v_0, -1)$ and $(1, 0)$ flow to (j, r) in R.
- If $h > 1$, then for all $i \neq j \in V$, there is some r_i such that (i, r_i) flows to (j, r) in R and i can reach m-height $h - 1$ by round r_i in R.

Thus the only difference between the m-height and height definitions is in the condition required to reach m-height 1. In the case of m-height we not only require that j has heard the input but also that j has heard from process 1. We also define $ML_i^r(R)$, $ML_i(R)$, $ML(R)$ analogously to the previous definitions for L_i.

Because of the small difference in the definitions, it is easy to show that the modified level measure differs by at most one from the level measure. Also the modified level measured by any two processes can differ by at most one.

Lemma 6.1 For all R and $i \in V$, $L_i(R) - 1 \leq ML_i(R) \leq L_i(R)$.

Lemma 6.2 For all R and $i, j \in V$, $ML_j(R) \geq ML_i(R) - 1$.

We will design a protocol based closely on the lower bound arguments of the previous section. Recall that we had shown that for any F, $\mathcal{L}(F, R) \leq \epsilon L(R)$. We have also seen that the modified level measure differs by at most one from the level measure. Thus in order to come close to meeting the lower bound, we will design a protocol in which:

- Each process i will calculate $ML_i(R)$, the value of the modified level at the end of the current run R.
- Each process will decide to attack with a probability proportional to $ML_i(R)$. This causes the liveness of the protocol to grow with $ML_i(R)$.

To do so each process i in protocol S has a variable $count_i$ that counts the value of $ML_i^r(R)$. We say that i has begun counting if $count_i > 0$. We will see how i begins counting below. However, once i has begun counting, process i increases $count_i$ to s (for $s > 1$) when it has heard that all other processes have reached a count of $s-1$. It is easy to implement this if each message sent by a node i carries $count_i$. and a variable called $seen_i$, the set of nodes that i knows has reached $count_i$.

Protocol S must satisfy agreement with parameter ϵ. Let $t = 1/\epsilon$. Process 1 chooses a random number *rfire* uniformly distributed in the range $(0, t]$ and passes it on all messages. After N rounds, i decides to attack if i has heard the value of *rfire* from process 1 and $count_i \geq rfire$.

Process i starts counting (i.e., sets $count_i$ to 1) in round r as soon it finds out that $(v_0, -1)$ and $(1, 0)$ flows to (i, r). We have discussed the reason for the second condition. The first condition, of course, is imposed to ensure validity. To implement the first condition, we use a variable $valid_i$ at each process i that is set to *true* in the first round r such that $(v_0, -1)$ flows to (i, r). To implement the second condition, all processes other than process 1 initially set the value of $rfire_i$ to a special value *undefined* which is updated when a message is received with the value of *rfire*.

6.1 Protocol Code

Protocol S consists of local state machines, each of which has a set of states, an initial state, a state transition function, a message generation function, and an output decision function. We describe each component in turn:

Each process i has the following state variables:

- $count_i$: integer between 1 and N (counts the value of $ML_i^r(R)$ in the current run R.).
- $rfire_i$: either a default value of *undefined* or a real number in the range $(0, 1/\epsilon]$. We assume that the value of *undefined* is not in $(0, 1/\epsilon]$.
- $seen_i$: a subset of V (represents the processes that have reached $count_i$ that i knows about).
- $valid_i$: a boolean (that is true if i has heard from v_0.)

We also use three temporary variables at each process: $highcount_i$ (an integer), $highseen_i$ (a subset of V), and $highset_i$ (a set of messages, whose format we describe later.)

The initial states are as follows. Process 1 initially sets $rfire_1$ to a a random number uniformly distributed in the range $(0, 1/\epsilon]$. All processes i other than 1, set $rfire_i$ = *undefined*. The $valid_i$ bit is only set if process i has received an input message from v_0 in Round 0. Finally process 1 sets $count_1 = 1$ iff $valid_1 = 1$. All other processes i initially set $count_i = 0$.

A message is denoted by m and has fields $m(rfire)$, $m(count)$, $m(seen)$, and $m(valid)$. The message generation function for i in every round sends a message $m(rfire, count, seen, valid)$ to all neighbors with $m(rfire) = rfire_i$, $m(count) = count_i$, $m(seen) = seen_i$, $m(valid) = valid_i$. Thus i sends a message with its current state to all neighbors in every round.

At the end of a round r, for $1 \leq r \leq N$, process i executes the procedure PROCESS-MESSAGE(S_i, i) where S_i is the set of messages process i has received in round r. PROCESS-MESSAGE(S_i, i) is shown in Figure 1. The first four lines are used to decide when a process starts counting; the remainder of the code does the actual counting.

Finally at the end of N rounds, $O_i = 1$ iff $rfire_i \neq$ undefined and $count_i \geq rfire_i$.

6.2 Proof of Properties of Protocol S

Notation: Consider any execution $Ex(R, \alpha)$. Let v^r denote the value of a variable at the end of round r. For example, $count_i^r$ denotes the value of $count_i$ at the end of r rounds. Define *rfire* to be the value of $rfire_1$ in the initial state.

Our first major step will be to establish that $count_i^r = ML_i^r(R)$. To allow a careful inductive proof,

```
PROCESS-MESSAGE(S_i, i)

If (rfire_i = undefined) and (∃m ∈ S_i :
   m(rfire) ≠ undefined) then rfire_i := m(rfire)
If (valid_i = false) and (∃m ∈ S_i : m(valid) = true)
   then valid_i := true
If (valid_i = true) and (rfire_i ≠ undefined)
   and (count_i = 0) then count_i := 1

If (count_i ≥ 1) and (S_i ≠ ∅) then
   highcount := Max_{m∈S_i} m(count)
   highset := {m ∈ S_i : m(count) = highcount}
   highseen := ∪_{m∈highset} m(seen)
   If highcount = count_i then
      seen_i := seen_i ∪ highseen ∪ {i}
   Else
      If highcount > count_i then
         seen_i := highseen ∪ {i};
         count_i := highcount;
   If seen_i = V then
      count_i := count_i + 1;
      seen_i := {i};
```

Figure 1: Procedure executed by process i at the end of a round in Protocol S

we will introduce invariants. The invariants should be intuitively clear from the previous discussion. The proofs of these invariants are deferred to the final paper.

Lemma 6.3 For any execution $Ex(R,\alpha)$ of Protocol S, the following assertions are true for $0 \leq r \leq N$ and for all $i,j \in V$:

1. $rfire_i^r$ is either equal to *rfire* or *undefined*.
2. $count_i^r \geq 1$ iff $rfire_i^r = rfire$ and $valid_i^r = true$.
3. $(1,0)$ flows to (i,r) iff $rfire_i^r = rfire$.
4. $(v_0,-1)$ flows to (i,r) iff $valid_i^r = true$.
5. If (j,s) flows to (i,r) in R then either $(count_i^r > count_j^s)$ or $(j \in seen_i^r$ and $count_i^r = count_j^s)$ or $(count_i^r = count_j^s = 0)$.
6. If $(j \in seen_i^r)$ then there is some s such that $(count_j^s = count_i^r)$ and (j,s) flows to (i,r) in R.
7. $seen_i^r \neq V$ and $seen_i^r \neq V - \{i\}$. Also, if $count_i^r \geq 1$ then $i \in seen_i^r$.
8. $ML_i^r \geq count_i^r$.

These invariants can now be used to establish that each process counts a value equal to its modified level measure. This should not be hard to believe since the code follows the definition of modified level.

Lemma 6.4 For all $i \in V$, any r such that $0 \leq r \leq N$, and any execution $Ex(R,\alpha)$ of Protocol S: $count_i^r = ML_i^r(R)$.

Proof: From the last invariant in Lemma 6.3, we see that $count_i^r \leq ML_i^r(R)$. So we show that $count_i^r \geq ML_i^r(R)$. We do so by induction on the value of $ML_i^r(R)$.

First if $ML_i^r(R) = 0$ we are done trivially since $count_i^r$ is always nonnegative. We use $ML_i^r(R) = 1$ as the base case. Then from the definition of $ML_i^r(R)$, we know that $(v_0,-1)$ and $(1,0)$ flow to (i,r) in run R. Hence by the third and fourth invariants in Lemma 6.3, $rfire_i^r = rfire$ and $valid_i^r = true$. Hence by the second invariant in Lemma 6.3, $count_i^r \geq 1$.

Next, suppose $ML_i^r(R) = l > 1$. Then from the definition of $ML_i^r(R)$, we know that for all $j \neq i$ there exists $r_j < r$ such that (j,r_j) flows to (i,r) in run R and $ML_j^{r_j} = l-1$. Hence by the fifth invariant in Lemma 6.3 and the inductive hypothesis, either $count_i^r > l-1$ (in which case we are done) or for all $j \neq i$, $j \in seen_i^r$. But the second case contradicts the seventh invariant in Lemma 6.3, and so we are done. □

Next we sketch proofs of the validity, unsafety, and liveness properties of S.

Theorem 6.5 Protocol S satisfies validity.

Proof: Informally, in any execution in which no process receives an input signal, no process hears from v_0, and so $count_i^N = 0$ for all i. Thus by the output decision function, $O_i = 0$ for all i in this execution.

More formally, fix a run R such that $I(R) = \{\}$, a random vector α, and any process i. Consider the execution $Ex(R,\alpha)$. Thus $(v_0,-1)$ does not flow to (i,N) for any $i \in V$. Thus by Invariant 4 in Lemma 6.3, $valid_i^N = false$. Hence by Lemma 6.3, Invariant 2, $count_i^N < 1$. Hence $count_i^N = 0$. However, $rfire_i^N$ by Invariant 1, Lemma 6.3, is either equal to *rfire* (which is strictly greater than 0) or *undefined*. In either case, by the output decision function, $O_i = 0$ in $Ex(R,\alpha)$. □

To prove the unsafety and liveness properties of S we characterize when the total attack and no attack events occur. Let *Mincount* be the minimum across all processes i of the value of $count_i$ at the end of an execution. The next lemma states that all processes

will attack if *Mincount* is no less than *rfire*, and no process will attack if *Mincount* is strictly less than *rfire* − 1;

Lemma 6.6 Fix an execution E of Protocol S. If $Mincount \geq rfire$ then $E \in TA$; but if $Mincount < rfire - 1$ then $E \in NA$.

Proof: If $Mincount \geq rfire$ then for all processes i, $count_i^N \geq rfire$. But $rfire > 0$, hence for all i, $count_i^N \geq 1$. Hence (by Lemma 6.3, invariant 2), for all i, $rfire_i^N = rfire$. Hence for all i, $count_i^N \geq rfire_i^N$ and $rfire_i^N \neq undefined$. Hence for all i, (by the decision function), $O_i = 1$. Hence, $E \in TA$.

If $Mincount < rfire - 1$, then using Lemma 6.4 and using the fact that the modified level measured at any two processes differs by at most 1 (Lemma 6.2), for all i, $count_i^N < rfire$. Now (by Lemma 6.3, Invariant 1), either $rfire_i^N = rfire$ or $rfire_i^N = undefined$. Hence, for all $i \in V$, either $count_i^N < rfire_i^N$ or $rfire_i^N = undefined$. Thus by the definition of the output decision function $O_i = 0$ for all i. Hence $E \in NA$. □

Theorem 6.7 S satisfies agreement with parameter ϵ.

Proof: By definition $U_s(S)$ is the maximum across all runs R of $Pr[PA|R]$. Consider any execution $E = Ex(R, \alpha)$. Now partial attack PA is the complement of the no attack and total attack events, NA and TA. From Lemma 6.6, we know that either TA or NA will occur unless $Mincount < rfire \leq Mincount+1$. Hence $Pr[PA|R] \leq Pr[Mincount < rfire \leq Mincount+1|R]$. Now for a given R, *Mincount* is fixed while *rfire* is a uniformly distributed random number in the range $(0, 1/\epsilon]$. Thus $U_s(s) \leq \epsilon$. □

Theorem 6.8 $\mathcal{L}(S, R) \geq Min(1, \epsilon ML(R))$.

Proof: Recall the definition of $\mathcal{L}(S, R)$ as the probability of total attack, $Pr[TA|R]$. We find a lower bound on $Pr[TA|R]$. Consider any execution E. From Lemma 6.6, $E \in TA$ if $Mincount \geq rfire$. But by Lemma 6.4 and the definition of *Mincount*, $Mincount = ML(R)$. Hence, $E \in TA$ if $ML(R) \geq rfire$. Thus for any run R, $Pr[TA|R]$ is no less than $Pr[ML(R) \geq rfire|R]$. Now for a given R, $ML(R)$ is fixed while *rfire* is a uniformly distributed random number in the range $(0, 1/\epsilon]$. Thus $Pr[TA|R]$ is no less than $Min(1, \epsilon ML(R))$. □

7 Closing the Gap: A Second Lower Bound

Theorem 5.4 states that for every run R and every protocol F, the liveness $\mathcal{L}(F, R)$ of any protocol F is at most $Min(1, \epsilon L(R))$. We described a protocol S whose liveness is $Min(1, \epsilon ML(R))$. From Lemma 6.1, we know that $ML(R)$ differs from $L(R)$ by at most one. Thus we have a small but irritating gap of ϵ. Our second lower bound shows, under a reasonable set of conditions that we call the usual case assumption, that no protocol F can do better than $\epsilon ML(R)$ on all runs R. More precisely, if any protocol F has a run R such that $\mathcal{L}(F, R) > \epsilon ML(R)$ then there is some other run $\tilde{R}$ such that $\mathcal{L}(F, \tilde{R}) < \epsilon ML(\tilde{R})$. Thus together the two bounds show that Protocol S is indeed "optimal".

A precise description of the second lower bound is in the appendix. We note that the proof of the first lower bound is similar to the chain arguments used often in deterministic impossibility results (e.g., [FL]). However, in proving the second lower bound, we are led to some connections between causality, probabilistic independence, and probabilistic agreement that may be interesting in their own right.

8 Conclusions

A strong adversary can be used to model a situation where links can crash and restart at an arbitrary frequency. A solution to coordinated attack is important in situations where consensus must be reached across unreliable links and within a specified time constraint. For coordinated attack against a strong adversary, we have seen that no protocol can achieve a tradeoff between liveness and safety ($\mathcal{L}/U$) that is better than linear in the number of rounds. This is bad news. For example if we want to achieve liveness with probability 1 on some run, and yet limit the probability of error to be less than 0.001, then the protocol must run for at least 1000 rounds. Protocol S demonstrates that the lower bounds are tight, but its performance is far from adequate. While our results are stated in a synchronous model, it seems clear that they can be extended to an asynchronous model.

In practice, there are two approaches that may help us to overcome these limitations. One approach is to add redundant links and assume that failures can only affect some fraction of the links in the network; then solutions similar to Byzantine Agreement can be used. However, this approach is expensive. The other approach is to assume a weaker failure model

than a strong adversary. One such adversary, which we call a *weak adversary*, is a *probabilistic* adversary which can destroy messages with a probability p that is not known in advance. We have preliminary results that show vastly improved performance against such an adversary.

9 Acknowledgements

We are grateful to Hagit Attiya, Baruch Awerbuch, Mihir Bellare, Cynthia Dwork, Ken Goldman, Steve Ponzio and Larry Stockmeyer for their help and suggestions.

References

[1] M. Ben-Or. Another advantage of free choice. *Proceedings 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30. August 1983

[2] J. Gray. Notes on Data Base Operating Systems. *Technical Report, IBM Report RJ2183(30001)*, IBM, February 1978, pp. 291–302, 1989. (Also in Operating Systems: An advanced course, Springer-Verlag Lecture Notes in Computer Science No. 60.)

[3] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Proceedings of the 3rd Annual Symposium on Principles of Distributed Computing*, pages 50-61, 1984.

[4] L. Lamport. Time Clocks and the ordering of events in a distributed System. *Communications of the ACM*, 21(7): 558-565, 1977.

[5] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3): 382-401, July 1981.

[6] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133-138, 1981.

A Lower Bound Based on Independence

Our second lower bound needs the following assumption. We say that the ***usual case assumption*** holds if:

- The graph G is connected and the diameter of G is no more than the number of rounds N.
- $\epsilon < 0.5$.

It is easy to see that these two conditions capture the usual and interesting cases. If the first condition does not hold then it can be shown that $L_i(R) \leq 1$ for all i, R, F and so by Lemma 5.4, $\mathcal{L}(F, R) \leq \epsilon$. Similarly if the second condition does not hold, the protocol is allowed to fail more than half the time. Thus the conditions preclude parameter settings that force absurdly small values of liveness and allow absurdly large values of unsafety.

Theorem A.1 Under the usual case assumption, if any protocol F has a run R such that $\mathcal{L}(F, R) > \epsilon ML(R)$ then there is some other run $\tilde{R}$ such that $\mathcal{L}(F, \tilde{R}) < \epsilon ML(\tilde{R})$.

The proof exploits a simple connection between probabilistic independence and what we call causal independence. Intuitively, two processes are causally independent if there is no causal flow, possibly through another process, that can link the two processes. For any $i, j \in V$, we say that i and j are ***causally independent*** in run R if there is no $k \in V$ such that $(k, 0)$ flows to (i, N) and $(k, 0)$ flows to (j, N) in R. The connection is expressed by the intuitive lemma:

Lemma A.2 If i and j are causally independent in run R then the events $(D_i|R)$ and $(D_j|R)$ are independent events.

If i and j are causally independent in run R, then there must be some restrictions on their decision probabilities in R in order to preserve the agreement property. There are several ways in which these restriction can be phrased; we select one that is sufficient for the later development.

Lemma A.3 Consider a run R in which i and j are causally independent and such that $Pr[D_i|R] = \epsilon$. Then if $\epsilon < .5$, $Pr[D_j|R] = 0$.

Proof: Let $Pr[D_j|R] = \delta$. We know that $Pr[PA|R] \geq Pr[D_i\overline{D_j}|R] + Pr[D_j\overline{D_i}|R]$. But since i and j are causally independent in R we have by Lemma A.2

that the events $(D_i|R)$ and $(D_j|R)$ are independent. Hence, $Pr[PA|R] \geq \epsilon(1-\delta) + \delta(1-\epsilon)$ and so $Pr[PA|R] \geq \epsilon + \delta(1-2\epsilon)$. But since $\epsilon < 0.5$, $1 - 2\epsilon > 0$. Hence by agreement, $\delta = 0$. □

For the next lemma, recall the definition of $ML_i(R)$, the modified level of process i in run R. This lemma serves to set up the proof of the following lemma, Lemma A.5.

Lemma A.4 Suppose that for all runs R and for all $i \in V$, $Pr[D_i|R] = 0$ if $ML_i(R) = 0$. Then for all R and $i \in V$, $Pr[D_i|R] \leq ML_i(R)\epsilon$.

Proof: By induction on the value of $ML_i(R)$. Let $ML_i(R) = l$.

Base Case, $l = 0$: This is the assumption of the lemma.

Inductive Step, $l > 0$: Using a lemma similar to Lemma 5.2, we can show that if $\tilde{R} = Clip_i(R)$, then there is some k such that $ML_k(\tilde{R}) = l - 1$. Hence by inductive assumption, $Pr[D_k|\tilde{R}] \leq \epsilon(l-1)$. Hence by Lemma 2.2, $Pr[D_i|\tilde{R}] \leq \epsilon l$. But by Lemma 4.2 and Lemma 2.1, $Pr[D_i|R] = Pr[D_i|\tilde{R}] \leq \epsilon l$. □

Now consider a run R_1 in which only process 1 receives an input message and no other message is delivered in the run. The next lemma states that if the probability of process 1 attacking in this run is exactly ϵ, then we can prove a tighter lower bound on the decision probabilities than the bound of Lemma 5.3. Recall that the bound in Lemma 5.3 was stated in terms of $L_i(R)$.

Lemma A.5 Suppose that $R_1 = \{(v_0, 1, 0)\}$, $Pr[D_1|R_1] = \epsilon$ and $\epsilon < 0.5$. Then for all runs R and all $i \in V$, $Pr[D_i|R] \leq ML_i(R)\epsilon$.

Proof: Consider any i and any R such that $ML_i(R) = 0$. Then we will claim that $Pr[D_i|R] = 0$. To do this we consider two cases, one of which must be true if $ML_i(R) = 0$.

- $(v_0, -1)$ does not flow to (i, N) in R. Then $L_i(R) = 0$ and hence by Lemma 5.3, $Pr[D_i|R] = 0$.
- $(1, 0)$ does not flow to (i, N) in R. Thus $i \neq 1$ as $(1, 0)$ flows to $(1, N)$. Consider the run $Clip_i(R)$. By the definition of clipping, there is no tuple $(*, 1, *)$ in $Clip_i(R)$, because if there was, $(1, 0)$ would flow to (i, N) in R. Consider the run $\tilde{R} = Clip_i(R) \cup \{(v_0, 1, 0)\}$. By construction, the only tuple of the form $(*, 1, *)$ in $\tilde{R}$ is $(v_0, 1, 0)$. Hence 1 and i are causally independent in $\tilde{R}$.

  Also, $R_1 = Clip_1(\tilde{R})$ and hence $R_1 \stackrel{1}{\equiv} \tilde{R}$. Thus $Pr[D_1|\tilde{R}] = \epsilon$. Hence by Lemma A.3, $Pr[D_i|\tilde{R}] = 0$. But $Clip_i(\tilde{R}) = Clip_i(R)$ and so by Lemma 4.2 and Lemma 2.1, $Pr[D_i|R] = Pr[D_i|\tilde{R}] = 0$.

Thus in either case, we have shown that for any i and R, $Pr[D_i|R] = 0$ if $ML_i(R) = 0$. The lemma now follows from Lemma A.4. □

Lemma A.6 Suppose the graph G is connected and has diameter no more than N. Then there is a run R such that $ML_1(R) = ML(R) = 1$, and the only tuple of the form $(*, 1, *)$ is $(v_0, 1, 0)$.

Proof: Let T be a spanning tree of G with 1 as the root. Such a tree exists because G is connected. Next we define R as follows.

- $I(R) = \{(v_0, 1, 0)\}$ (i.e., only process 1 receives input).
- For all $i, j \in V$ and $1 \leq r \leq N$, $(i, j, r) \in R$ iff i is the parent of j in the tree. (i.e., information only flows down the tree.)

It is not hard to see that since the height of the tree is no more than N, $ML_1(R) = 1$ and $ML_i(R) \geq 1$ for all $i \in V$. Thus $ML(R) = 1$. □

We now return to the proof of Theorem A.1.

Proof: Suppose there is some protocol F such that for all R, $\mathcal{L}(F, R) \geq \epsilon ML(R)$.

By Lemma A.6, there is a run R_1 such that $ML_1(R_1) = ML(R_1) = 1$ and the only tuple of the form $(*, 1, *)$ in R_1 is $(v_0, 1, 0)$. It is easy to verify that $L_1(R_1) = 1$.

Thus by assumption, $\mathcal{L}(S, R_1) \geq \epsilon ML(R_1) = \epsilon$. Thus by Lemma 2.3, $Pr[D_1|R_1] \geq \epsilon$. Also, by Lemma 5.3, since $L_1(R_1) = 1$, $Pr[D_1|R_1] \leq \epsilon$. Hence, $Pr[D_1|R_1] = \epsilon$.

Now consider the run $R_2 = Clip_1(R_1) = \{(v_0, 1, 0)\}$. Then by Lemma 4.2 $R_2 \stackrel{1}{\equiv} R_1$. Thus by Lemma 2.1, $Pr[D_1|R_2] = \epsilon$. Hence by Lemma A.5, for all i, R, $Pr[D_i|R] \leq \epsilon ML_i(R)$. Thus for all R, $Min_i Pr[D_i|R] \leq Min_i \epsilon ML_i(R)$. Thus from Lemma 2.3 and the definition of $ML(R)$, $\mathcal{L}(F, R) \leq \epsilon ML(R)$.

Thus we have shown that for any protocol F, if for all R, $\mathcal{L}(F, R) \geq \epsilon ML(R)$, then $\mathcal{L}(F, R) = \epsilon ML(R)$. This implies the theorem. □

Fast Randomized Algorithms for Distributed Edge Coloring *
(EXTENDED ABSTRACT)

Alessandro Panconesi & Aravind Srinivasan
Department of Computer Science, Cornell University
Ithaca, NY 14853
E-mail: {ap, srin}@cs.cornell.edu

Abstract

Certain types of routing, scheduling and resource allocation problems in a distributed setting can be modeled as edge coloring problems. We present fast and simple randomized algorithms for edge coloring a graph, in the synchronous distributed point–to–point model of computation. Our algorithms compute an edge–coloring of a graph G with n nodes and maximum degree Δ with at most $(1.6+\epsilon)\Delta + \log^{2+\delta} n$ colors with high probability (arbitrarily close to 1), for any fixed $\epsilon, \delta > 0$.

To analyze the performance of our algorithms, we introduce new techniques for proving upper bounds on the tail probabilities of certain random variables. *Chernoff–Hoeffding bounds* are fundamental tools that are used very frequently in estimating tail probabilities. However, they assume stochastic independence among certain random variables, which may not always hold. Our results extend the Chernoff–Hoeffding bounds to certain types of random variables which are not stochastically independent. We believe that these results are of independent interest, and merit further study.

1 Introduction

An important limitation for a distributed network without global memory in computing any function is that for an efficient algorithm, each processor can communicate only with those processors that are within a small radius around it. Models of parallel computation like the PRAM abstract this problem of locality away by assuming the existence of a global shared memory with fast concurrent access. We are interested in studying how fast individual processors can compute their portion of the output in a message–passing distributed system, with such "local" information alone. The model we study is the synchronous distributed point–to–point model, in which the processors are arranged as the vertices of an n–vertex graph $G = (V, E)$, and where all communication is via the edges of G alone.

We present fast and simple randomized algorithms to edge color G with at most $(1.6+\epsilon)\Delta + 0.4\log^{2+\delta} n$ colors with high probability for any fixed $\epsilon, \delta > 0$, where Δ is the maximum degree of the vertices of G. At the heart of our analysis is an extension of the Chernoff–Hoeffding bounds, which are key tools in bounding the tail probabilities of certain random variables.

The edge coloring problem can be used to model certain types of jobshop scheduling, packet routing, and resource allocation problems in a distributed setting. For example, given a set of processes $\mathcal{P}$ and a set of resources $\mathcal{R}$ such that each process $p \in \mathcal{P}$ needs a subset $f(p) \subseteq \mathcal{R}$ of the resources where: (i) each process p needs every resource in $f(p)$ for a unit of time each, and (ii) p can use the resources in $f(p)$ in any order, we can construct a bipartite graph $G_{\mathcal{P},\mathcal{R}} = (\mathcal{P}, \mathcal{R}, E_{\mathcal{P},\mathcal{R}})$ where $E_{\mathcal{P},\mathcal{R}} = \{(p,r) | p \in \mathcal{P} \wedge r \in f(p)\}$, and an edge

*This research was supported in part by NSF PYI award CCR–89–96272 with matching support from UPS and Sun Microsystems.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0251...$1.50

coloring of $G_{\mathcal{P},\mathcal{R}}$ with c colors yields a schedule for the processes to use the resources within c time units.

Edge coloring can also be used in distributed models in situations where broadcasts are infeasible or undesirable: an edge coloring of the network results in a schedule for each processor to communicate with at most one neighbor at every step (at time step i, processors communicate via the edges colored i only), and using a "small" number of colors reduces the wastage of time in this schedule.

Note that Δ colors are necessary to edge color a graph with maximum degree Δ. Vizing showed that it is always possible to edge color a graph with $\Delta + 1$ colors and gave a polynomial time algorithm [7]. Karloff & Shmoys gave an RNC algorithm for this problem that uses $\Delta + \Delta^{0.5+\epsilon}$ colors, which was derandomized in NC by Berger & Rompel, and Motwani, Naor & Naor [6, 14]. In the distributed model, the best known edge coloring algorithm is to apply a vertex coloring algorithm to the line graph of G. There are fast (polylogarithmic) randomized vertex coloring algorithms that use $(\Delta+1)$ and Δ colors, which translate to $(2\Delta - 1)$- and $(2\Delta - 2)$-edge coloring algorithms respectively [13, 15]. There are no known polylogarithmic deterministic algorithms in the distributed setting for $(2\Delta - 1)$–edge coloring [3, 15]. Moreover, distributed Δ–edge coloring requires $\Omega(diameter(G))$ time, even with randomization[15].

This work is part of a larger research where we are interested in studying the problem of a distributed model solving some problem (related to scheduling and synchronization) on its own topology. The main problems that have been studied in this regard are computing a maximal independent set (MIS) in G [1, 12], a $(\Delta+1)$-vertex coloring of G [3, 13], and a Δ–vertex coloring of G [15]. An MIS captures the idea of a set of processors working in parallel without interfering with the decisions made by their neighbors, and a vertex coloring is a partition of the processors into independent sets, yielding a schedule for the processors to work in parallel.

An important point about all of these problems is that they need just a *local search* for an *incremental update*, in the following sense. Consider the simple sequential algorithm to compute an MIS– pick any vertex v, add it to the partial MIS computed so far, remove v and its neighbors from G, and repeat. Thus, a partial MIS can be updated incrementally with a local search of radius 1. Similar "local search" results for incremental updates are known for $(\Delta + 1)$- and Δ–vertex coloring (and hence for $(2\Delta - 1)$- and $(2\Delta - 2)$–edge coloring): local searches of radius 1 and radius $O(\log_\Delta n)$ respectively [15]. Hence, the key problem in implementing these in a parallel setting is *symmetry breaking*: parallelizing the incremental sequential algorithm (implied by such a local result) by somehow breaking the symmetrical effort of all the processors to execute one incremental step of the sequential algorithm. Two key tools for symmetry breaking are randomization [1, 12, 13] and *network decomposition* [2, 3, 4, 5, 11, 15].

However, such a local search result is not known for edge coloring with less than $2\Delta - 2$ colors. Note that if we are allowed $2\Delta-1$ colors, an uncolored edge is always surrounded by at most $2\Delta-2$ colors, and a local search of radius 1 is sufficient for the update. It is not clear that any local result should hold when we have at most $2\Delta - 3$ colors. Hence, one main contribution of this paper is a fast and simple distributed algorithm for a problem not falling under the hitherto dominant paradigm of "symmetry breaking and local search". Our algorithms are also very simple; we first present two algorithms for edge coloring bipartite graphs, and then extend them to general graphs using an idea from [10]. A sketch of our first algorithm: given a bipartite graph $G = (A, B, E)$ with maximum degree Δ, each vertex in B picks distinct colors from $\{1, 2, \ldots, \Delta\}$ at random for its edges; then, each vertex $v \in A$ checks, for each color α, if more than one of its incident edges has color α and if so, chooses one of them at random as the *winner*, and all the other edges of color α which are incident to v are decolored. The key claim, which requires an interesting analysis, is that for every vertex, the number of decolored edges incident to it is at most $\Delta(1 + \epsilon)/e$ with high probabil-

ity. Assuming that this holds, we can repeat the above iteration with a set of $\Delta(1+\epsilon)/e$ *fresh* colors, etc., and the above claim allows us to bound the number of colors used, with high probability.

The second main contribution of this paper is the tools developed to analyze the algorithms. *Chernoff-Hoeffding* (henceforth CH) bounds[8, 9] are fundamental tools used in bounding the tail probabilities of the sum of independent random variables [17]. The most frequent form in which these bounds are used is when there are n ***independent*** random bits $X_1, X_2, \ldots X_n$, $X = \sum_{i=1}^{n} X_i$, and $\mu = E[X]$; in this case, it is possible to show that $Pr(X \geq (1+\delta)\mu) \leq F^+(\mu, \delta)\ \forall \delta > 0$, where $F^+(\mu, \delta)$ is inverse exponential in μ and δ^2, for small enough δ [16, 17].

We introduce a new way of looking at the CH bounds and prove that CH type bounds can be used for the sums of certain types of dependent random variables too. A generalization to some form of dependence is known [18], but it is not strong enough to be used in our applications. We also prove similar results to certain types of dependent *non-binary* random variables. These new extensions of the CH bounds are used crucially in the analysis of our algorithms. We believe that these results have the potential for further applications, and need further study.

2 A Generalization of Chernoff-Hoeffding Bounds

Chernoff-Hoeffding (henceforth CH) bounds are important tools used in estimating the tail probabilities of random variables. Given n random variables $X_1, X_2, \ldots, X_n$, these bounds are used in deriving an upper bound on the upper tail probability $Pr(X \geq (1+\epsilon)\mu)$, where $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $\epsilon > 0$. Chernoff's basic idea for bounding the upper tail is to use Markov's inequality on the random variable e^{tX} for an arbitrary, but fixed, $t > 0$, and minimize with respect to t. That is, to use the fact that

$$Pr(X \geq (1+\epsilon)\mu) = Pr(e^{tX} \geq e^{t(1+\epsilon)\mu}) \leq \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu}},$$

and minimize the last ratio for $t > 0$ [8]. Raghavan and Spencer use this idea to bound the upper tail when $X_1, X_2, \ldots, X_n$ are independent random bits [16, 17], and show that in this case,

$$Pr(X \geq (1+\epsilon)\mu) \leq \left(\frac{e^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu} = F^+(\mu, \epsilon).$$

If ϵ is a fixed positive quantity and is at most 1 (which will be true in all our applications), then $F^+(\mu, \epsilon) \leq e^{-\epsilon^2\mu/3}$. If $\mu = \Omega(\log^{1+c} t)$ then $F^+(\mu, \epsilon)$ is asymptotically less than any inverse polynomial in t, for any fixed $c > 0$. This fact will be used in our algorithms.

Hoeffding also has derived a (slightly stronger) bound on the upper tail of the sum of independent 0-1 random variables [9]. Further, he has proved bounds on the upper tail of the sum of independent and bounded non–binary variables, and for the sum of certain types of bounded and mutually dependent random variables. In this section, we present upper bounds on the upper tails of some types of ***dependent*** random variables. We present an important special case first (Theorem 1) before showing the general result (Theorem 2).

2.1 Self-weakening binary random variables

Given n 0-1 random variables $X_1, X_2, \ldots, X_n$, we call them *self-weakening with parameter* λ if $Pr(X_{i_1} = X_{i_2} = \cdots = X_{i_k} = 1) \leq \lambda \cdot \prod_{j=1}^{k} Pr(X_{i_j} = 1)$ for all distinct indices $i_1, i_2, \ldots, i_k$ in the range $\{1, 2, \ldots, n\}$. Note that if $\lambda = 1$, a sufficient condition for this to hold is that $\forall i : 1 \leq i \leq n$, and $\forall (j_1, \ldots, j_k) : \{j_1, \ldots, j_k\} \subseteq \{1, 2, \ldots, i-1\}$, $Pr(X_i = 1 | X_{j_1} = X_{j_2} = \cdots = X_{j_k} = 1) \leq Pr(X_i = 1)$.

Theorem 1 *Consider n 0-1 random variables $X_1, X_2, \ldots, X_n$ which are self-weakening with parameter λ, for some $\lambda > 0$. Let $X = \sum_{i=1}^{n} X_i$, and let $E[X] \leq \mu^*$, for some μ^*. Then,*

$$Pr(X \geq (1+\epsilon)\mu^*) \leq \lambda \cdot F^+(\mu^*, \epsilon).$$

Note that any upper bound μ^* on $E[X]$ will do. This theorem is a special case of Theorem 2, which will be proved in subsection 2.2. We now present applications of Theorem 1, which will be used later and are also of independent interest.

Example 1 *Δ balls are thrown uniformly at random, and independently of each other, into Δ bins. Let X be the random variable denoting the resulting number of empty bins. Then, for any $\epsilon > 0$, $Pr(X \geq (1+\epsilon)\Delta/e) \leq F^+(\Delta/e, \epsilon)$.*

PROOF. Let X_i be the indicator variable for the event that bin i was empty after the experiment; note that $X = \sum_{i=1}^{\Delta} X_i$. It is easy to check that $E[X] \leq \Delta/e \equiv \mu^*$. Notice that the X_i's are dependent and hence we cannot apply the CH bounds directly. However, the X_i's are self-weakening with parameter 1 since, for any set of indices $\{j_1, j_2, \ldots, j_k\} \subseteq \{1, 2, \ldots, \Delta\} - \{i\}$, we have that $Pr(X_i = 1 | X_{j_1} = X_{j_2} = \cdots = X_{j_k} = 1) = (1 - 1/(\Delta - k))^{\Delta} \leq (1 - 1/\Delta)^{\Delta} = Pr(X_i = 1)$.

Hence, $Pr(X \geq (1+\epsilon)\Delta/e) \leq F^+(\Delta/e, \epsilon)$ follows from Theorem 1. □

Remark. Jain has proved the following lemma [18]: *Let $a_1, a_2, \ldots, a_n$ be n random trials (not necessarily independent) such that the probability that trial a_i 'succeeds' is bounded above by p regardless of the outcomes of the other trials. Then if X is the random variable that represents the number of 'successes' in these n trials, and Y is a binomial variable with parameters (n, p), then: $Pr[X \geq k] \leq Pr[Y \geq k], 0 \leq k \leq n$.*

The assumptions of Jain's lemma are strictly stronger than the self-weakening property with parameter 1; such strong assumptions do not hold in example 1. For instance, $Pr(X_\Delta = 1 | X_1 = X_2 = \cdots = X_{\Delta-1} = 0) = \frac{\Delta-1}{\Delta+1}$, which, for large enough Δ, is greater than $Pr(X_\Delta = 1) (\approx 1/e)$.

Another application of Theorem 1, which will also be used later in the analysis of our algorithms, is given in the following example.

Example 2 *Suppose $d \leq \Delta$ balls are thrown uniformly at random and independently into Δ bins, and in each bin with at least one ball, one of the balls from that bin is chosen at random, and the other balls in that bin are discarded. Denoting by Z_d the number of discarded balls, $Pr(Z_d \geq (1+\epsilon)\Delta/e) \leq F^+(\Delta/e, \epsilon)$.*

PROOF. Omitted. □

2.2 A new look at Chernoff-Hoeffding type bounds

Hoeffding[9] has shown that if n random variables $X_1, X_2, \ldots, X_n$ are independent with $a_i \leq X_i \leq b_i$ $(1 \leq i \leq n)$, and if $X = \sum_{i=1}^{n} X_i$ with $E[X] = \mu$, then for any $\epsilon > 0$,

$$\begin{aligned} Pr(X \geq (1+\epsilon)\mu) &\leq e^{-2\mu^2\epsilon^2/(\sum_{i=1}^{n}(b_i - a_i)^2)} \\ &= G^+(\mu, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n). \end{aligned}$$

Note that if $\forall i$, $b_i - a_i$ is bounded by some constant and if $\mu = \Theta(n)$, then $G^+(\mu, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n) = e^{-\Theta(n\epsilon^2)}$, which is asymptotically lesser than any inverse polynomial in n, for any fixed $\epsilon > 0$. We now present CH-bounds that hold for certain types of non-binary random variables, which satisfy a condition of which "self-weakening" is a special case.

Theorem 2 *Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a random variable such that $X_i \in [a_i, b_i]$. If $E[X] \leq \mu^*$ and there exists $\lambda > 0$ such that, for all nonempty sets of indices $I \subseteq \{1, \ldots, n\}$ and strictly positive integer values s_i, $E[\prod_{i \in I} X_i^{s_i}] \leq \lambda \cdot \prod_{i \in I} E[X_i^{s_i}]$, then $Pr(X > (1+\epsilon)\mu^*) \leq \lambda \cdot G^+(\mu^*, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$.*

PROOF. (SKETCH). Since X is bounded by assumption, it follows for any t (in particular, for any $t > 0$), that linearity of expectation holds for the infinite sum

$$E[e^{tX}] = E\left[\sum_{i=0}^{\infty} \frac{t^i X^i}{i!}\right] = \sum_{i=0}^{\infty} \frac{t^i}{i!} E[X^i].$$

$E[X^i]$ contains terms of the form $E[\prod_{j=1}^{n} X_j^{s_j}]$, which, by assumption, is at most $\lambda \cdot \prod_{j=1}^{n} E[X_j^{s_j}]$. Note that if $Y_1, \ldots, Y_n$ are *independent* random variables with Y_i having the same distribution as X_i, then $E[e^{tY}] = \prod_{i=1}^{n} E[e^{tY_i}]$, where $Y = \sum_{i=1}^{n} Y_i$, and also that for all non-negative integers $s_1, s_2, \ldots, s_n$, $E[\prod_{i=1}^{n} Y_i^{s_i}] = \prod_{i=1}^{n} E[Y_i^{s_i}] = \prod_{i=1}^{n} E[X_i^{s_i}]$. Hence, $E[e^{tX}] \leq \lambda \cdot \prod_{i=1}^{n} E[e^{tX_i}]$ and so for any $t > 0$,

$$Pr(X \geq (1+\epsilon)\mu^*) \leq \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu^*}} \leq \lambda \cdot \frac{\prod_{i=1}^{n} E[e^{tX_i}]}{e^{t(1+\epsilon)\mu^*}},$$

which can be bounded by $\lambda \cdot G^+(\mu^*, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$, by picking a suitable $t > 0$ [9]. □

A special case of Theorem 2 is given in the following corollary and should be of independent interest.

Corollary 1 *Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a non-negative, discrete valued random variable with values in $[a_i, b_i]$. If $E[X] \leq \mu^*$ and there exists a $\lambda > 0$ such that, for all nonempty sets of indices $I \subseteq \{1, \ldots, n\}$ and $c_i > 0$, $Pr[\bigwedge_{i \in I} X_i = c_i] \leq \lambda \cdot \prod_{i \in I} Pr[X_i = c_i]$, then $Pr(X > (1+\epsilon)\mu^*) \leq \lambda \cdot G^+(\mu^*, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$.*

PROOF. It is easily checked that the assumptions of this corollary are a special case of the assumptions of Theorem 2. □

Theorem 1 is actually a corollary of Theorem 2; the assumptions of Theorem 1 again are a special case of those of Theorem 2, and Theorem 1 follows from the proof of Theorem 2, and from the existence of a suitable $t > 0$ to make the upper tail bound at most $\lambda \cdot F^+(\mu^*, \epsilon)$ [16, 17].

The basic idea of Theorem 2 is that if we can upper-bound each term of the series expansion of $E[e^{tX}]$ with an equivalent term from the series expansion of $E[e^{tU}]$, where U is the sum of independent random variables U_i's whose range is the same as that of the X_i's, the CH-bounds of U apply also to the random variable X. With the same reasoning of Theorem 2 we can prove the following corollaries.

Corollary 2 *Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a random variable such that $X_i \in [a_i, b_i]$, and let $U = \sum_{i=1}^{n} U_i$ where the U_i's are independent random variables such that $U_i \in [a_i, b_i]$. If $E[X], E[U] \leq \mu^*$ and there exists $\lambda > 0$ such that, for all nonempty sets of indices $I \subseteq \{1, \ldots, n\}$ and strictly positive integer values s_i, $E[\prod_{i \in I} X_i^{s_i}] \leq \lambda \cdot \prod_{i \in I} E[U_i^{s_i}]$, then $Pr(X > (1+\epsilon)\mu^*) \leq \lambda \cdot G^+(\mu^*, \epsilon, a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$.*

Corollary 3 *Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a random variable such that $X_i \in [0, 1]$, and let $U = \sum_{i=1}^{n} U_i$, with the U_i's being independent binary random variables. If $E[X], E[U] \leq \mu^*$ and there exists $\lambda > 0$ such that, for all nonempty sets of indices $I \subseteq \{1, \ldots, n\}$ and strictly positive integer values s_i, $E[\prod_{i \in I} X_i^{s_i}] \leq \lambda \cdot \prod_{i \in I} E[U_i]$, then $Pr(X > (1+\epsilon)\mu^*) \leq \lambda \cdot F^+(\mu^*, \epsilon)$.*

Both corollaries will be used in the analysis of our algorithms. The next example contains an application of corollary 2. We first give a simple lemma without proof.

Lemma 1 *If $0 \leq p \leq 1$, $q = 1 - p$ and m is a positive integer, then*

$$\sum_{r=1}^{m} \binom{m}{r} p^r q^{m-r} \frac{r}{r+1} = 1 - \frac{(1 - q^{m+1})}{p(m+1)}.$$

Example 3 *Suppose d white balls have been thrown at random into Δ bins, $1 \leq d < \Delta$. After this, a red ball is thrown at random into one of the bins, one ball is chosen at random from the bin in which the red ball fell, and the other balls in that bin are discarded. Let Z be the random variable denoting the probability that the red ball is discarded as a function of the positions of the white balls (Z itself a random variable depending on the positions of the white balls). Then, $E[Z] \leq 1/e$ and $Pr(Z > (1+\epsilon)1/e) \leq e^{-2\epsilon^2 \Delta / e^2}$.*

PROOF. Let Z_i be the random variable denoting the number of white balls in bin i. Then, $Z = \Delta^{-1}\sum_{i=1}^{\Delta} Z_i/(Z_i+1)$. Let $Y_i = Z_i/(Z_i+1)$ and $Y = \sum Y_i$. Note that the Y_i's are discrete random variables with values in $[0,1]$ and that $Y = \Delta \cdot Z$. We will show that $E[Y] \leq \Delta/e$ and that $Pr(Y \geq (1+\epsilon)\Delta/e) \leq e^{-2\Delta\epsilon^2/e^2}$, which will give our claim.

Firstly, we may assume that $d = \Delta - 1$: $Pr(Y \geq (1+\epsilon)\Delta/e)$ is maximized at $d = \Delta - 1$, as d varies from 1 to $\Delta - 1$ (proof omitted). First, we will show that, for all i, $E[Y_i] \leq 1/e$ and then we will show that, for any set of k indices $I \subseteq \{1, 2, \ldots, \Delta\}$ and strictly positive integers s_i,

$$E[\prod_{i\in I} Y_i^{s_i}] \leq 1/e^k. \tag{1}$$

Given this we can apply Corollary 2 by introducing n independent 0-1 random variables U_i such that $E[U_i] = Pr(U_i = 1) = 1/e$. Since the U_i's are binary, equation 1 is the same as $E[\prod_{i\in I} Y_i^{s_i}] \leq \prod_{i\in I} E[U_i^{s_i}]$.

Noting that $0 \leq Y_i \leq 1$, it suffices to show that

$$E[\prod_{i\in I} Y_i] \leq 1/e^k. \tag{2}$$

Without loss of generality we can assume $I = \{1, \ldots, k\}$. We will prove inequality 2 by induction on $k \geq 1$; when $k = 1$,

$$\begin{aligned} E[Y_1] &= \sum_{r=0}^{\Delta-1} \binom{\Delta-1}{r} (1/\Delta)^r (1-1/\Delta)^{\Delta-1-r} \frac{r}{r+1} \\ &= (1-1/\Delta)^{\Delta} \leq 1/e, \end{aligned}$$

where the second equality follows from Lemma 1. Notice that for all j, $1 \leq j \leq \Delta$, $E[Y_j] = E[Y_1] \leq 1/e$. When $k > 1$, the law of conditional probabilities gives

$$E[\prod_{i=1}^{k} Y_i] = E[Y_1 Y_2 \cdots Y_{k-1} E[Y_k | Y_1, \ldots, Y_{k-1}]]. \tag{3}$$

If we show that, for all non-zero $c_i \in [0,1]$ with $i \in \{1, \ldots, k-1\}$,

$$E[Y_k | \bigwedge_{i=1}^{k-1} Y_i = c_i] \leq \frac{1}{e} \tag{4}$$

then, since if some c_i is zero then the product $Y_1 Y_2 \ldots Y_{k-1}$ is also zero, we see by induction on k that

$$\begin{aligned} E[\prod_{i=1}^{k} Y_i] &= E[Y_1 \ldots Y_{k-1} E[Y_k | Y_1 \ldots Y_{k-1}]] \\ &\leq \frac{1}{e} E[\prod_{i=1}^{k-1} Y_i] \\ &\leq \frac{1}{e^k}. \end{aligned}$$

Hence, Example 3 follows if we can show that inequality 4 holds.

If a_i denotes the number of white balls that fell into bin i, then $c_i = a_i/(a_i+1)$. Let $a = \sum_{i=1}^{k-1} a_i \geq k-1$, $p = 1/(\Delta - k + 1)$, and $q = 1 - p$. Then

$$\begin{aligned} E[Y_k | \bigwedge_{i=1}^{k-1} Y_i = c_i] &= E[Y_k | \bigwedge_{i=1}^{k-1} Z_i = a_i] \\ &= \sum_{r=1}^{\Delta-1-a} t(r,a), \end{aligned}$$

where

$$t(r,a) \doteq \binom{\Delta-1-a}{r} p^r q^{\Delta-1-a-r} \frac{r}{r+1}.$$

It is easy to check that $t(r,a) \geq t(r,a+1)$. As a consequence, the maximum value of $E[Y_k | \bigwedge_{i=1}^{k-1} Y_i = c_i]$ is attained at $a = k-1$, in which case we have

$$\begin{aligned} \sum_{r=1}^{\Delta-1-a} t(r,a) &= \sum_{r=1}^{\Delta-k} t(r,k-1) \\ &= \sum_{r=1}^{\Delta-k} \binom{\Delta-k}{r} p^r q^{\Delta-k-r} \frac{r}{r+1} \\ &= q^{\Delta-k+1} \leq 1/e, \end{aligned}$$

by Lemma 1. □

3 Edge Coloring Algorithms for Bipartite Graphs

We present two bipartite edge coloring algorithms now. In the next section, we will see how they can be used as subroutines to compute edge colorings of arbitrary graphs within the same bounds. We first present a simple distributed Monte Carlo algorithm for edge coloring bipartite graphs, and analyze its performance with the techniques developed in the preceding section. The algorithm takes $O(\log n)$ time and colors a bipartite graph G of degree Δ with approximately $1.6\Delta + 0.4\log^{2+\delta} n$ colors with high probability, for any $\delta > 0$ (the failure probability will depend on δ). We will say that a statement holds *with high probability* (w.h.p.) if the probability that it holds is at least $1 - 1/f(n)$, where $f(n)$ is asymptotically greater than any polynomial of n.

Given a bipartite graph $G = (A, B, E)$, we denote by $\delta(u)$ the set of edges incident on vertex u. Each vertex knows whether it belongs to A or B. This is an important assumption because it cannot be computed fast distributively [15], but it will be removed when we discuss edge coloring general graphs. We initialize a variable Δ_{curr} to Δ; during any iteration of the algorithm, Δ_{curr} is meant to be an upper bound on the degree of the current graph; we will prove later that this holds w.h.p. The algorithm is given two parameters $\epsilon, \delta > 0$, and is as follows:

1. PART I: Repeat until $\Delta_{curr} < \log^{2+\delta} n$:
   Let G' be the current graph. Pick a set χ of Δ_{curr} fresh new colors.
   - *(Assigning temporary colors)* In parallel and independently of the other vertices in B, each vertex $v \in B$ assigns a temporary color to each edge in $\delta(v)$ with uniform probability *without replacement, i.e.* edge e_1 is assigned color $\alpha \in \chi$ with probability $1/\Delta_{curr}$, e_2 is assigned $\beta \in \chi - \{\alpha\}$ with probability $1/(\Delta_{curr} - 1)$ and so on.
   - *(Choosing winners)* The coloring so far is consistent around any vertex $v \in B$ but can be inconsistent around a vertex $u \in A$. For each $u \in A$, let $C_u(\alpha)$ be the set of edges in $\delta(u)$ with temporary color α. Each vertex $u \in A$ selects a winner uniformly at random in $C_u(\alpha)$, for each nonempty $C_u(\alpha)$. All other edges are decolored and assigned color $\perp$.
   - Set $\Delta_{curr} := \Delta_{curr}(1+\epsilon)/e$. $G_\perp$, the subgraph of G' induced by the $\perp$-edges, becomes the new current graph.
2. PART II: Let G_r be the remaining graph. Edge color G_r with $2\Delta(G_r) - 1$ colors by executing Luby's vertex coloring algorithm on the line graph of G_r for $O(\log n)$ time [13].

To bound the number of colors used, we will prove that the maximum degree of the graph shrinks by a factor of at least $(1+\epsilon)/e$ w.h.p. in every iteration of part (I) above, *i.e.*, that in every iteration, Δ_{curr} is an upper bound for the degree of the graph at the start of that iteration. This would imply that the maximum degree of G_r is at most $\log^{2+\delta} n$ w.h.p. Hence, w.h.p., the number of colors used is at most

$$C = \Delta + \frac{\Delta}{e}(1+\epsilon) + \ldots + \frac{\Delta}{e^k}(1+\epsilon)^k + 2\log^{2+\delta} n,$$

where k is the smallest integer such that $\Delta(1+\epsilon)^k/e^k \leq \log^{2+\delta} n$. The total number of colors C is upper bounded by

$$\begin{aligned} C &\leq (\frac{e}{e-1} + \epsilon')\Delta + (2 - \frac{e}{e-1} - \epsilon')\log^{2+\delta} n \\ &\approx 1.6\Delta + 0.4\log^{2+\delta} n. \end{aligned}$$

Here, ϵ' depends on ϵ and can be made arbitrarily small. The running time of the algorithm is $O(\log n)$: Part I takes $O(\log \Delta)$ time and Part II, i.e., Luby's algorithm, takes $O(\log n)$ time.

Consider any iteration of part (I) above, with an upper bound Δ_{curr} on the degree of the graph at the beginning of that iteration. For each edge e, we introduce

an indicator random variable H_e that is 1 when e gets color $\perp$ in that iteration, and 0 otherwise (if e has a vertex v as an endpoint, we also denote H_e by $H_v(e)$). When $H_e = 1$ we say that edge e is *hit*. The variable $H_v = \sum_{e \in \delta(v)} H_e$ equals the number of edges incident to v that are hit, and represents the degree of v in the next iteration. It is easy to show that, for each vertex v, $E[H_v] \leq \Delta_{curr}/e$. But, we also need to estimate the tail probability of the distribution of H_v. With the techniques developed in the previous section, we will be able to prove CH-bounds for the upper tail of H_v and this will be sufficient for our purposes, since if μ denotes Δ_{curr}/e, then

$$\begin{aligned} Pr(\Delta_\perp > (1+\epsilon)\mu) &= Pr(\exists v : H_v > (1+\epsilon)\mu) \\ &\leq \sum_{v \in A \cup B} Pr(H_v > (1+\epsilon)\mu) \\ &\leq n \cdot c \cdot \Delta_{curr} \cdot F^+(\sqrt{\mu/e}, \epsilon/2), \end{aligned}$$

where $\Delta_\perp$ is the new degree of the graph after that iteration, $c \cdot \Delta_{curr} \cdot F^+(\sqrt{\mu/e}, \epsilon/2)$ is the upper bound that we will prove on the upper tail of the H_v's, and c is a constant. Henceforth, $\sqrt{\Delta_{curr}}$ will be denoted by Δ_0. The following lemma is easy:

Lemma 2 *In any iteration and for any edge (u,v), $Pr((u,v)$ is hit$) \leq e^{-1}$.*

Hence, given that Δ_{curr} was an upper bound on the degree of the graph at the start of that iteration, $E[H_v] \leq \Delta_{curr}/e$, for all vertices v. We now want to prove

Theorem 3 *In any one iteration, $Pr(H_v > (1+\epsilon)\Delta_{curr}/e) \leq c \cdot \Delta_{curr} \cdot F^+(\Delta_0/e, \epsilon/2)$ for all vertices v, given that Δ_{curr} was an upper bound on the degree of the graph at the start of that iteration.*

The proof of this theorem will take several lemmas. Note that $F^+(\Delta_0/e, \epsilon/2) \leq \exp(-\epsilon^2\Delta_0/12e)$ if $\epsilon \leq 2$; this is asymptotically less than any inverse polynomial in n when $\Delta_{curr} = \Omega(\log^{2+\delta} n)$ for any fixed $\delta > 0$, if ϵ is fixed. Hence, if theorem 3 is true, then Δ_{curr} is an upper bound on the maximum degree in *every* iteration w.h.p. When Δ_{curr} is smaller than $\log^{2+\delta} n$, the probability bound is not good any more, and so we switch to Part (II) of the algorithm then.

Henceforth, we focus on any one iteration, assuming that Δ_{curr} was an upper bound on the maximum degree at the start of that iteration. It turns out that proving Theorem 3 is much simpler for the A vertices than for the B vertices. We start by analyzing the vertices in A. Given some vertex $u \in A$, we want to upper bound the upper tail of H_u. We show now that for the edges $e \in \delta(u)$, the random variables H_e, though dependent, are self-weakening with parameter 1. Note that, as far as $u \in A$ is concerned, each of the edges in $\delta(u)$ chooses a color independently at random with probability Δ_{curr}^{-1}; hence the situation is analogous to example 2. Hence, by example 2,

Lemma 3 *For any vertex $v \in A$, $Pr(H_v > (1+\epsilon)\Delta_{curr}/e) \leq F^+(\Delta_{curr}/e, \epsilon)$.*

The derivation of such a result for vertices in B is much harder. The problem can be seen in Figure 1. Suppose we are given that e_1 got temporary color α and was hit, and that e_2 got temporary color β; we will argue intuitively that given this, the probability of e_2 being hit has increased. Since e_1 was hit, the probability of e_3 getting temporary color α increases, which implies that the probability of e_4 getting temporary color β also increases, and this increases the probability of e_2 being hit. So, it is not obvious that, for $v \in B$, the $H_v(e)$'s are self-weakening; in fact, the above argument seems to imply the antithesis of that. With a series of lemmas, we will prove that they are self-weakening (with parameter to be determined later), and Theorem 3 will follow. We will again focus on any particular iteration, at the start of which Δ_{curr} was an upper bound on the maximum degree of the graph. We consider some vertex $v \in B$, and assume that its degree is Δ_{curr} (easily extended to the case $degree(v) < \Delta_{curr}$).

Let v's neighbors in A be $u_1, u_2, \ldots, u_{\Delta_{curr}}$, and denote the edge (v, u_i) by e_i. For technical reasons that

will be clear later, we subdivide the edges $\delta(v)$ into Δ_0 groups (recall that Δ_0 denotes $\sqrt{\Delta_{curr}}$), such that each group has Δ_0 edges. Let H_v^i be the random variable counting the number of edges of $\delta(v)$ in group i that are hit. Then, $Pr(H_v > (1+\epsilon)\Delta_{curr}/e) \leq Pr(\exists i, 1 \leq i \leq \Delta_0 : H_v^i > (1+\epsilon)\Delta_0/e) \leq \sum_{i=1}^{\Delta_0} Pr(H_v^i > (1+\epsilon)\Delta_0/e)$. We will prove that the generalized CH bounds hold for any of the H_v^i's and get

$$\begin{aligned} Pr(H_v > (1+\epsilon)\Delta_{curr}/e) &\leq \Delta_0 Pr(H > (1+\epsilon)\Delta_0/e) \\ &\leq c \cdot \Delta_{curr} \cdot F^+(\Delta_0/e, \epsilon/2) \end{aligned}$$

where, to simplify our notation, H stands for the sum of any choice of at most Δ_0 random variables chosen among the $H_v(e_i)$s, say, $H_v(e_1), H_v(e_2), \ldots, H_v(e_{\Delta_0})$.

To estimate the tail probability of H, we assume that v was the last among the vertices in B to choose its permutation, *i.e.*, that the edges $e_1 = (v, u_1), \ldots, e_{\Delta_{curr}} = (v, u_{\Delta_{curr}})$ got their temporary colors after all other edges incident to the u_i's had got temporary colors. Define, for each u_i, a column vector C_i with Δ_{curr} entries, where $C_i(j) = a/(a+1)$ iff u_i has exactly a edges with temporary color j before the random choice of edge e_i.

To prove that the $H_v(e_i)$'s are self-weakening, we analyze the generic term $Pr(H_v(e_{i_1}) = \cdots = H_v(e_{i_k}) = 1)$, where $k \leq \Delta_0$ and each i_j is in the range $1, 2, \ldots, \Delta_0$. Construct a $\Delta_{curr} \times k$ matrix M by concatenating $C_{i_1}, \ldots, C_{i_k}$. M contains the information for computing $Pr(H_1 = H_2 \cdots = H_k = 1)$ (where H_j denotes $H_v(e_{i_j})$, $1 \leq j \leq k$), and this will be our next step. Though M is not square, we define its *permanent* $perm(M)$ in the obvious way: $perm(M) = \sum \prod_{j=1}^{k} M(a_j, j)$, where the sum is taken over all the a_js in the range $\{1, 2, \ldots \Delta_{curr}\}$ such that $a_i \neq a_j$ for $i \neq j$. In what follows, let $p(n,k) = n(n-1)\ldots(n-k+1)$.

Lemma 4 *$Pr(H_1 = H_2 \cdots = H_k = 1) = perm(M)/p(\Delta_{curr}, k)$.*

Let $S(C_i)$ be the sum of the entries of C_i, E_1 be the event "for all i, $1 \leq i \leq \Delta_0$, $S(C_i)$ is at most $\Delta_{curr}(1+\epsilon')/e$" (for some $\epsilon' > 0$ to be specified), and E_2 be the event "$H > (1+\epsilon)\Delta_0/e$". In example 3 we showed that, for any C_i, $Pr(S(C_i) > (1+\epsilon')\Delta_{curr}/e) \leq e^{-2\Delta_{curr}\epsilon'^2/e^2}$ ($S(C_i)$ corresponds to the variable Y of Example 3); it follows that $Pr(E_1^c) \leq \Delta_0 \cdot e^{-2\Delta_{curr}\epsilon'^2/e^2}$.

It is not difficult to prove that $perm(M)$ is at most the product of the column sums in M, hence Lemma 4 implies that *given* E_1, $perm(M) \leq \Delta_{curr}^k (1+\epsilon')^k/(e^k p(\Delta_{curr}, k))$. Next, a simple lemma:

Lemma 5 *For positive integers t and k, $t^k/p(t,k) \leq e^{k^2/t}$, if $k \leq t/2$.*

By applying Lemma 5 with $t = \Delta_{curr}$ and $k \leq \Delta_0 = \sqrt{\Delta_{curr}}$, we have that **given** E_1,

$$perm(M) \leq e \cdot (1+\epsilon')^k/e^k \leq e \cdot e^{\epsilon'\Delta_0}/e^k. \quad (5)$$

Inequality 5 tells us that $Pr(H_v(e_{j_1}) = H_v(e_{j_2}) = \cdots = H_v(e_{j_k}) = 1 | E_1) \leq e \cdot e^{\epsilon'\Delta_0} \cdot Pr(X_{j_1} = X_{j_2} = \cdots = X_{j_k} = 1)$, where $\{j_1, j_2, \ldots, j_k\} \subseteq \{1, 2, \ldots, \Delta_0\}$, and $X_1, X_2, \ldots, X_{\Delta_0}$ are independent random bits with $Pr(X_i = 1) = 1/e$, $1 \leq i \leq \Delta_0$. Hence, by Corollary 3 we see that $Pr(E_2|E_1) \leq e \cdot e^{\epsilon'\Delta_0} \cdot F^+(\Delta_0/e, \epsilon)$. Thus,

$$\begin{aligned} Pr(E_2) &= Pr(E_2 \mid E_1)Pr(E_1) + Pr(E_2 \mid E_1^c)Pr(E_1^c) \\ &\leq Pr(E_2 \mid E_1) + Pr(E_1^c) \\ &\leq e \cdot e^{\epsilon'\Delta_0} \cdot F^+(\Delta_0/e, \epsilon) + \Delta_0 \cdot e^{-2\Delta_{curr}\epsilon'^2/e^2} \\ &\leq c \cdot \Delta_0 \cdot F^+(\Delta_0/e, \epsilon/2), \end{aligned}$$

for some constant c and for large enough Δ_0, by choosing $\epsilon' = \epsilon^2/(6e)$.

Lemma 6 *For all $v \in B$ and any iteration at the start of which Δ_{curr} was an upper bound on the maximum degree of the graph, $Pr(H_v > (1+\epsilon)\Delta_{curr}/e) \leq c \cdot \Delta_{curr} F^+(\sqrt{\Delta_{curr}}/e, \epsilon/2)$.*

This concludes our proof of Theorem 3. We now give a brief sketch of another bipartite edge coloring algorithm. The idea of the algorithm is to repeatedly remove a randomly generated matching and to make it

a color class. The matching is generated so that edges incident on "high" degree vertices are removed. This ensures that at each step the degree goes down by at least one. A nice property of this algorithm is that if Δ is "slightly more" than $\Omega(\log n)$ (to be quantified later), then after $O(\Delta)$ stages the diameter of any remaining connected component of the graph is $O(\log n)$. This allows us to find an optimal coloring of each component C by brute–force: elect a leader, which will then obtain complete information about C and compute an optimal edge coloring of C. We conjecture that this property is true of the previous algorithm also, but the analysis is complicated by the dependency among the edges. Recall that Luby's algorithm is invoked in Part (II) of the previous algorithm when the degree of the graph is $O(\log^{2+\delta} n)$; at this point we can use the new algorithm, instead of Luby's, without changing the the polylogarithmic complexity of the whole algorithm.

A brief sketch of the second algorithm follows. The algorithm takes $O(\Delta + \log n)$ time, but uses at most $(e/(e-1)+\epsilon)\Delta + 0.4\log^{1+2\delta} n$ colors w.h.p., if $\Delta \geq \log^{1+2\delta} n$ ($\delta > 0$). Denote $\Delta - (\frac{e-1}{e})(1-\epsilon)\log^{1+\delta} n$ by $f(\Delta, \epsilon, \delta)$. Given an upper bound Δ on the maximum degree of the graph and $\epsilon, \delta > 0$, a Δ–*phase* is as follows:

> *Repeat* $\log^{1+\delta} n$ *times:* Every vertex $v \in B$ with degree at least $f(\Delta, \epsilon, \delta)$ picks a random edge in $\delta(u)$ as a *candidate*. Now, every vertex $u \in A$ picks one of the candidate edges incident to it (if any) at random as a *winner*. The same fresh color is given to the winners, which are now deleted from the graph.

The edge coloring algorithm:
(1) Repeat until $\Delta \leq \log^{1+2\delta} n$:

- Execute a Δ–phase.
- Replace Δ by $f(\Delta, \epsilon, \delta)$.

(2) Use Luby's algorithm to color the remaining graph.

The key claim is that if $\Delta \geq \log^{1+2\delta} n$ and is an upper bound on the maximum degree of the graph, then w.h.p., the maximum degree of the graph after executing a Δ–phase is at most $f(\Delta, \epsilon, \delta)$. Also, if we start with a graph G where $\Delta(G) \geq \log^{1+2\delta+\delta'} n$ with $\delta' > 0$, then the diameter of any connected component of the graph remaining after Step (1) above is at most $O(\log n)$ w.h.p., and hence the brute–force approach can be used in Step (2) above, instead of Luby's algorithm.

4 Edge Coloring General Graphs

In this section, we give a brief sketch of a Monte Carlo distributed algorithm for edge coloring general graphs. Several technical details have been omitted for conciseness, and will appear in the final version of this paper. The algorithm is a recursive procedure based on an idea of Karloff & Shmoys [10], and uses our bipartite edge coloring algorithm as a subroutine. It can be applied on any graph with maximum degree $\Delta \geq \log^2 n$. It runs in $O(\log n)$ time, and uses at most $(e/(e-1)+\epsilon)\Delta + (2-(e/(e-1)-\epsilon))\log^{2+\delta} n$ colors w.h.p., for any given $\epsilon, \delta > 0$. Next, we state without proof that if $\Delta \geq \log^{4+\delta'} n$ for some $\delta' > 0$, then the algorithm can be made to use at most $(e/(e-1)+\epsilon)\Delta$ colors w.h.p. (the failure probability will depend on δ', but will be asymptotically less than any inverse polynomial in n, for any fixed $\delta' > 0$).

The idea of the algorithm, as in [10], is to first compute a random partition of the vertices of $G = (V, E)$ into black and white vertices, with each vertex deciding to go to one of the two subsets independently by the toss of a coin. Next, we color the bipartite graph induced by the edges with endpoints of different colors using our algorithm (note that *every vertex knows which side of the bipartite graph it is in*), and then recurse on the two disconnected induced subgraphs (the one induced by black vertices and the one induced by white vertices) using the same sample of fresh new colors on both. The key idea is that the maximum degree of each of these three subgraphs is at most $\Delta/2 + \Delta^{0.5+\epsilon}$ w.h.p., for any $\epsilon > 0$.

Assuming that this "degree-halving" is true, the maximum degree becomes $\log^{2+\delta} n$ in $O(\log \Delta)$ iterations at which point, we can apply Luby's algorithm to the resulting graphs G_r to get $(2\Delta(G_r) - 1)$ edge colorings of these graphs.

In fact, we can do better, if the initial degree $\Delta \geq \log^{4+\delta'} n$ for some $\delta' > 0$. A consequence of the Karloff & Shmoys partition idea is that the diameters of the induced subgraphs of the white and black vertices shrink at an exponential rate w.h.p., and hence after $O(\log \Delta)$ iterations, it becomes $O(\log n)$ w.h.p. (proofs omitted). This allows us to apply a brute- force search and color the finally remaining graphs G_r optimally (with $\Delta(G_r)$ or $\Delta(G_r) + 1$ colors, instead of $2\Delta(G_r) - 1$ colors). Hence, the number of colors used in this case is at most $(e/(e-1) + \epsilon)\Delta$, w.h.p. Though this is not a big reduction in the number of colors used, we think that this "diameter shrinking" is an important property of the Karloff & Shmoys algorithm, and should be useful in other contexts.

Acknowledgments

Our sincere thanks go to David Shmoys, for his continued guidance and support. We are very grateful to Éva Tardos for an important idea about analyzing the first edge coloring algorithm, to Pankaj Rohatgi with whom the second bipartite edge coloring algorithm was jointly developed, and to Stephen Vavasis for his useful suggestions. We are also grateful to Suresh Chari, Devdatt Dubhashi, David Pearson and Desh Ranjan for useful discussions.

References

[1] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1986.

[2] B. Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32:804–823, 1985.

[3] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 364–369, 1989.

[4] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM J. on Discrete Mathematics*, 5(2), 1992.

[5] B. Berger and L. Cowen. Fast deterministic constructions of low-diameter network decompositions. MIT-LCS Technical Memo #460, April 1991.

[6] B. Berger and J. Rompel. Simulating $(\log^c n)$-wise independence in NC. *J. Assoc. Comput. Mach.*, 38(4):1026–1046, 1991.

[7] B. Bollobás. *Graph Theory*. Springer Verlag, New York, 1979.

[8] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

[9] W. Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, pages 13–30, 1963.

[10] H. J. Karloff and D. B. Shmoys. Efficient parallel algorithms for edge coloring problems. *Journal of Algorithms*, 8:39–52, 1987.

[11] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proceedings of the ACM/SIAM Symposium on Discrete Algorithms*, pages 320–330, 1991.

[12] M. Luby. A fast parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[13] M. Luby. Removing randomness in parallel computation without a processor penalty. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 162–173, 1988.

[14] R. Motwani, J. Naor, and M. Naor. The probabilistic method yields deterministic parallel algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 8–13, 1989.

[15] A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *ACM Symposium on Theory of Computing*, pages 581–592, 1992.

[16] P. Raghavan. *Randomized Rounding and Discrete Ham-Sandwich Theorems: Provably Good Algorithms for Routing and Packing Problems.* PhD thesis, University of California at Berkeley, July 1986. Also available as Computer Science Department Report UCB/CSD 87/312.

[17] P. Raghavan. Lecture notes on randomized algorithms. Technical Report RC 15340 (#68237), IBM T.J.Watson Research Center, January 1990. Also available as CS661 Lecture Notes, Technical report YALE/DCS/RR-757, Department of Computer Science, Yale University, January 1990.

[18] R. Raman. The power of Collision: Randomized parallel algorithms for chaining and integer sorting. In *Proceedings, 10th Annual FST & TCS Conference*, Lecture Notes in Computer Science # 472, pages 161–175. Springer-Verlag, Berlin, December 1990. Also available as University of Rochester CS Dept. TR 336, March 1990 (Revised January 1991).

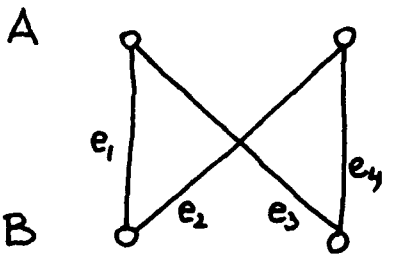


Figure 1: A case of bad dependence

Proving Probabilistic Correctness Statements: the Case of Rabin's Algorithm for Mutual Exclusion*

Isaac Saias†

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

The correctness of most randomized distributed algorithms is expressed by a statement of the form "some predicate of the executions holds with high probability, regardless of the order in which actions are scheduled". In this paper, we present a general methodology to prove correctness statements of such randomized algorithms. Specifically, we show how to prove such statements by a series of refinements, which terminate in a statement independent of the schedule. To demonstrate the subtlety of the issues involved in this type of analysis, we focus on Rabin's randomized distributed algorithm for mutual exclusion [6].

Surprisingly, it turns out that the algorithm does not maintain one of the requirements of the problem under a certain schedule. In particular, we give a schedule under which a set of processes can suffer lockout for arbitrary long periods.

1 Introduction

1.1 General Considerations

For many distributed system problems, it is possible to produce randomized algorithms that are better than their deterministic counterparts: they may be more efficient, have simpler structure, and even achieve correctness properties that deterministic algorithms cannot. One cost of using randomization is the increased difficulty of proving correctness of the resulting algorithms. A randomized algorithm typically involves two different types of nondeterminism – that arising from the random choices and that arising from an *adversary*. The interaction between these two kinds of nondeterminism complicates the analysis of the algorithm.

In the distributed system model considered here, each of a set of concurrent processes executes its local code and communicates with the others through a shared variable. The code can contain random choices, which leads to probabilistic branch points in the tree of executions. By assumption, the algorithm is provided at certain points of the execution with random inputs having known distributions. We can equivalently consider that *all* random choices made in a single execution are given by a parameter ω at the onset of the execution. The parameter ω thus captures the first type of nondeterminism.

For the second type, we here define the adversary $\mathcal{A}$ to be the entity controlling the order in which processes take steps. (In other work (e.g., [4]), the adversary can control other decisions, such as the contents of some messages.) An adversary $\mathcal{A}$ bases its choices on the *knowledge* it holds about the prior execution of the system. This knowledge varies according to the specifications for each given problem. In this paper, we will consider an adversary allowed to observe only certain "external" manifestations of the execution and having no access, for example, to information about local process states. We will say that an adversary is *admissible* to emphasize its specificity.

These two sources of nondeterminism, ω and $\mathcal{A}$, uniquely define an execution $\mathcal{E} = \mathcal{E}(\omega, \mathcal{A})$ of the algorithm.

Among the correctness properties one often wishes to prove for randomized algorithms are properties that state that a certain property W of executions has a "high" probability of holding against all admissible adversaries. Note that the probability men-

*Research supported by research contracts ONR-N00014-91-J-1046, NSF-CCR-8915206 and DARPA-N00014-89-J-1988.

†e-mail: saias@theory.lcs.mit.edu

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0263...$1.50

tioned in this statement is taken with respect to a probability distribution on executions. One of the major sources of complication is that there are two probability spaces that need to be considered: the space of random inputs ω and the space of random executions. Let $\mathbf{dP}$ denote the probability measure given for the space of random inputs ω.

Since the evolution of the system is determined both by the (random) choices expressed by ω and also by the adversary $\mathcal{A}$, we do not have a single probability distribution on the space of all executions. Rather, for each adversary $\mathcal{A}$ there is a corresponding distribution $\mathbf{dP}_{\mathcal{A}}$ on the executions "compatible with" $\mathcal{A}$. High probability correctness properties of a randomized algorithm $\mathcal{C}$ are then generally stated in terms of the distributions $dP_{\mathcal{A}}$, in the following form. Let W and I be sets of executions of $\mathcal{C}$ and let l be a real number in $[0,1]$. Then $\mathcal{C}$ is correct provided that $\mathbf{P}_{\mathcal{A}}[W \mid I] \geq l$ for every admissible adversary $\mathcal{A}$. For a condition expressed in this form, we think of W as the set of "good" (or "winning") executions, while I is a set that expresses the assumptions under which the good behavior is supposed to hold.

In general, it is difficult to calculate (good bounds on) probabilities of the form $P_{\mathcal{A}}[W|I]$. This is because the probability that the execution is in W, I or $W \cap I$ depends on a combination of the choices in ω and those made by the adversary. Although we assume a basic probability distribution $\mathbf{P}$ for ω, the adversary's choices are determined in a more complicated way – in terms of certain kinds of knowledge of the prior execution. In particular, the adversary's choices can depend on the outcomes of prior random choices made by the processes.

The situation is much simpler in the special case where the events W and I are defined directly in terms of the choices in ω. In this case, the desired probability can be calculated just by using the assumed probability distribution $\mathbf{dP}$.

Our general methodology for proving a high probability correctness property of the form $P_{\mathcal{A}}[W|I]$ consists of proving successive lower bounds:

$$\begin{aligned} \mathbf{P}_{\mathcal{A}}[W \mid I] &\geq \mathbf{P}_{\mathcal{A}}[W_1 \mid I_1] \\ &\vdots \\ &\geq \mathbf{P}_{\mathcal{A}}[W_r \mid I_r], \end{aligned}$$

where all the W_i and I_i are sets of executions, and where the last two sets, W_r and I_r, are defined directly in terms of the choices in ω. The final term, $\mathbf{P}_{\mathcal{A}}[W_r \mid I_r]$, is then evaluated (or bounded from below) using the distribution $\mathbf{dP}$. This methodology can be difficult to implement as it involves disentangling the ways in which the random choices made by the processes affect the choices made by the adversary.

This paper is devoted to emphasizing the need of such a rigorous methodology in correctness proofs: in the context of randomized algorithms the power of the adversary is generally hard to analyze and imprecise arguments can easily lead to incorrect statements.

As evidence supporting our point, we give an analysis of Rabin's randomized distributed algorithm [6] implementing mutual exclusion for n processes using a read-modify-write primitive on a shared variable with $O(\log n)$ values. Rabin claimed that the algorithm satisfies the following correctness property: for every adversary, any process competing for entrance to the critical section succeeds with probability $\Omega(1/m)$, where m is the number of competing processes. As we shall see, this property can be expressed in the general form $\mathbf{P}_{\mathcal{A}}[W \mid I] \geq l$. In [5], Sharir et al. gave another analysis of the algorithm, providing a formal model in terms of Markov chains; however, they did not make explicit the influence of the adversary on the probability distribution on executions.

We show that this influence is crucial: the adversary in [6] is much stronger than previously thought, and in fact, the high probability correctness result claimed in [6] does not hold.

1.2 Rabin's Algorithm

The problem of *mutual exclusion* [2] involves allocating an indivisible, reusable resource among n competing processes. A mutual exclusion algorithm is said to guarantee *progress*[1] if it continues to allocate the resource as long as at least one process is requesting it. It guarantees *no-lockout* if every process that requests the resource eventually receives it. A mutual exclusion algorithm satisfies *bounded waiting* if there is a fixed upper bound on the number of times any competing process can be bypassed by any other process. In conjunction with the progress property, the bounded waiting property implies the no-lockout property. In 1982, Burns et al.[1] considered the mutual exclusion algorithm in a distributed setting where processes communicate through a shared read-modify-write variable. For this setting, they proved that any *deterministic* mutual exclusion algorithm that guarantees progress and bounded waiting requires that the shared variable take on at least n distinct values. Shortly thereafter, Rabin published a *randomized* mutual exclusion algorithm [6] for the same shared memory distributed setting. His algorithm guarantees progress using a shared variable that takes on only $O(\log n)$ values.

It is quite easy to verify that Rabin's algorithm

[1] We give more formal definitions of these properties in Section 2.

guarantees mutual exclusion and progress; in addition, however, Rabin claimed that his algorithm satisfies the following informally-stated *strong no-lockout* property[2].

> *"If process i participates in a trying round of a run of a computation by the protocol and compatible with the adversary, together with* $0 \leq m-1 < n$ *other processes, then the probability that i enters the critical region at the end of that round is at least* c/m, $c \sim 2/3$." (*)

This property says that the algorithm guarantees an approximately equal chance of success to all processes that compete at the given round. Rabin argued in [6] that a good randomized mutual exclusion algorithm should satisfy this strong no-lockout property, and in particular, that the probability of each process succeeding should depend inversely on m, the number of *actual* competitors at the given round. This dependence on m was claimed to be an important advantage of this algorithm over another algorithm developed by Ben-Or (also described in [6]); Ben-Or's algorithm is claimed to satisfy a weaker no-lockout property in which the probability of success is approximately c/n, where n is the total number of processes, i.e., the number of *potential* competitors.

Rabin's algorithm uses a randomly-chosen round number to conduct a competition for each round. Within each round, competing processes choose lottery numbers randomly, according to a truncated geometric distribution. One of the processes drawing the largest lottery number for the round wins. Thus, randomness is used in two ways in this algorithm: for choosing the round numbers and choosing the lottery numbers. The detailed code for this algorithm appears in Figure 1.

We begin our analysis by presenting three different formal versions of the no-lockout property. These three statements are of the form discussed in the introduction and give lower bounds on the (conditional) probability that a participating process wins the current round of competition. They differ by the nature of the events involved in the conditioning and by the values of the lower bounds.

Described in this formal style, the strong no-lockout property claimed by Rabin involves conditioning over m, the number of participating processes in the round. We show in Theorem 3.1 that the adversary can use this fact in a simple way to lock out any process during any round.

On the other hand, the *weak* c/n no-lockout property that was claimed for Ben-Or's algorithm involves only conditioning over events that describe the knowledge of the adversary at the end of previous round. We show in Theorems 3.2 and 3.4 that the algorithm suffers from a different flaw which bars it from satisfying even this property.

We discuss here informally the meaning of this result. The idea in the design of the algorithm was to incorporate a mathematical procedure within a distributed context. This procedure allows one to select with high probability a unique random element from any set of at most n elements. It does so in an efficient way using a distribution of small support ("small" means here $O(\log n)$) and is very similar to the approximate counting procedure of [3]. The mutual exclusion problem in a distributed system is also about selecting a unique element: specifically the problem is to select in *each* trying round a unique process among a set of competing processes. In order to use the mathematical procedure for this end and select a true random participating process at each round and for *all choices of the adversary*, it is necessary to discard the old values left in the local variables by previous calls of the procedure. (If not, the adversary could take advantage of the existing values.) For this, another use of randomness was designed so that, with high probability, at each new round, all the participating processes would erase their old values when taking a step.

Our results demonstrate that this use of randomness did not actually fulfill its purpose and that the adversary is able in some instances to use old lottery values and defeat the algorithm.

In Theorem 3.5 we show that the two flaws revealed by our Theorems 3.1 and 3.2 are at the center of the problem: if one restricts attention to executions where program variables are reset, and if we disallow the adversary to use the strategy revealed by Theorem 3.1 then the strong bound does hold. Our proof highlights the general difficulties encountered in our methodology when attempting to disentangle the probabilities from the influence of $\mathcal{A}$.

The algorithm of Ben-Or which is presented at the end of [6] is a modification of Rabin's algorithm that uses a shared variable of *constant* size. All the methods that we develop in the analysis of Rabin's algorithm apply to this algorithm and establish that Ben-Or's algorithm is similarly flawed and does not satisfy the $1/2en$ no-lockout property claimed for it in [6]. Actually, in this setting, the shared variables can take only two values, which allows the adversary to lock out processes with probability one, as we show

[2] In the statement of this property, a "trying round" refers to the interval between two successive allocations of the resource, and the "critical region" refers to the interval during which a particular process has the resource allocated to it. A "critical region" is also called a "critical section".

in Theorem 3.8.

In a recent paper [7], Kushilevitz and Rabin use our results to produce a modification of the algorithm, solving randomized mutual exclusion with $\log_2{}^2 n$ values. They solve the problem revealed by our Theorem 3.1 by conducting *before* round k the competition that results in the control of *Crit* by the end of round k. And they solve the problem revealed by our Theorem 3.2 by enforcing in the code that the program variables are reset to 0.

The remainder of this paper is organized as follows. Section 2 contains a description of the mutual exclusion problem and formal definitions of the strong and weak no-lockout properties. Section 3 contains our results about the no-lockout properties for Rabin's algorithm. It contains Theorems 3.1 and 3.2 which disprove in different ways the strong and weak no-lockout properties and Theorem 3.5 whose proof is is a model for our methodology: a careful analysis of this proof reveals exactly the origin of the flaws stated in the two previous theorems. One of the uses of randomness in the algorithm was to disallow the adversary from knowing the value of the program variables. Our Theorems 3.2 and 3.7 express that this objective is not reached and that the adversary is able to infer (partially) the value of *all* the fields of the shared variable. Theorem 3.8 deals about the simpler setting of Ben-Or's algorithm.

Some mathematical properties needed for the constructions of Section 3 are presented in an appendix (Section 4).

2 The Mutual Exclusion Problem

The problem of *mutual exclusion* is that of continually arbitrating the exclusive ownership of a resource among a set of competing processes. The set of competing processes is taken from a universe of size n and changes with time. A solution to this problem is a distributed algorithm described by a program (code) $\mathcal{C}$ having the following properties. All involved processes run the same program $\mathcal{C}$. $\mathcal{C}$ is partitioned into four regions, *Try*, *Crit*, *Exit*, and *Rem* which are run cyclically in this order by all processes executing $\mathcal{C}$. A process in *Crit* is said to hold the resource. The indivisible property of the resource means that at any point of an execution, at most one process should be in *Crit*.

2.1 Definition of Runs, Rounds, and Adversaries

In this subsection, we define the notions of *run*, *round*, *adversary*, and *fair* adversary which we will use to define the properties of *progress* and *no-lockout*.

A *run* ρ of a (partial) execution $\mathcal{E}$ is a sequence of triplets $\{(p_1, old_1, new_1), (p_2, old_2, new_2), \ldots (p_t, old_t, new_t) \ldots\}$ indicating that process p_t takes the t^{th} step in $\mathcal{E}$ and undergoes the region change $old_t \rightarrow new_t$ during this step (e.g., $old_t = new_t = Try$ or $old_t = Try$ and $new_t = Crit$). We say that $\mathcal{E}$ is *compatible* with ρ.

An admissible *adversary* for the mutual exclusion problem is a mapping $\mathcal{A}$ from the set of finite runs to the set $\{1, \ldots, n\}$ that determines which process takes its next step as a function of the current partial run. That is, the adversary is *only* allowed to see the changes of regions. For every t and for every run $\rho = \{(p_1, old_1, new_1), (p_2, old_2, new_2), \ldots\}$, $\mathcal{A}[\{(p_1, old_1, new_1), \ldots, (p_t, old_t, new_t)\}] = p_{t+1}$. We then say that ρ and $\mathcal{A}$ are *compatible*.

An adversary $\mathcal{A}$ is *fair* if for every execution, every process i in *Try*, *Crit*, or *Exit* is eventually provided by $\mathcal{A}$ with a step. This condition describes "normal" executions of the algorithm and says that processes can quit the competition only in *Rem*.

A *round* of an execution is the part between two successive entrances to the critical section (or before the first entrance). Formally, it is a maximal execution fragment of the given execution, containing one transition $Try \rightarrow Crit$ at the end of this fragment and no other transition $Try \rightarrow Crit$. The round of a run is defined similarly.

A process i *participates* in a round if i takes a step while being in its trying section *Try*.

2.2 The Progress and No-Lockout Properties

Definition 2.1 An algorithm $\mathcal{C}$ that solves mutual exclusion guarantees *progress* if, for all fair adversaries, there is no infinite execution in which, from some point on, at least one process is in its *Try* region (respectively its *Exit* region) and no transition $Try \rightarrow Crit$ (respectively $Exit \rightarrow Rem$) occurs.

The properties that we considered thus far are non-probabilistic. The no-lockout property is probabilistic. Its formal definition requires the following notation:

Let X denote any generic quantity whose value changes as the execution unfolds (e.g., a program variable). We let $X(k)$ denote the value of X *just prior* to the last step ($Try \rightarrow Crit$) of the kth round of the execution. As a special case of this general notation, we define the following.

$\mathcal{P}(k)$ is the set of participating processes in round k. (Set $\mathcal{P}(k) = \emptyset$ if $\mathcal{E}$ has fewer then k rounds.) The notation $\mathcal{P}(k)$ is consistent with the general notation because the set of processes participating in round k is updated as round k progresses: in effect the definition of this set is complete only at the end of round k (this fact is at the heart of our Theorem 3.1).

$t(k)$ is the total number of steps that are taken by all the processes up to the end of round k.

$\mathcal{N}(k)$ is the set of executions in which all the processes j participating in round k reinitialize their program variables B_j with a new value $\beta_j(k)$ during round k. ($\mathcal{N}$ stands for New-values.) $\beta_j(k)$; $k = 1, 2, \ldots$, $j = 1, \ldots, n$ is a family of iid [3] random variable whose distribution is geometric truncated at $\log_2 n + 4$ (see [6]).

For each i and k, we let $W_i(k)$ denote the set of executions in which process i enters the critical region at the end of round k.

We consistently use the probability theory convention according to which, for any property $\mathcal{S}$, the set of executions $\{\mathcal{E} : \mathcal{E}$ has property $\mathcal{S}\}$ is denoted as $\{\mathcal{S}\}$. Then:

- For each step number t and each execution $\mathcal{E}$ we let $\pi_t(\mathcal{E})$ denote the run compatible with the first t steps of $\mathcal{E}$. For any t-steps run ρ, $\{\pi_t = \rho\}$ represents the set of executions compatible with ρ. ($\{\pi_t = \rho\} = \emptyset$ if ρ has fewer then t steps.) We will use π_k in place of $\pi_{t(k)}$ to simplify notation.
- Similarly, for all $m \leq n$, $\{|\mathcal{P}(k)| = m\}$ represents the set of executions having m processes participating in round k.

The quantities $\mathcal{N}(k), \{\pi_t = \rho\}$, $W_i(k)$, $\{|\mathcal{P}(k)| = m\}$ are sets of executions: for a given adversary they are random events in the probability space of random executions endowed with the measure $\mathbf{dP}_{\mathcal{A}}$.

We now present the various no-lockout properties that we want to study. A first question is to characterize relevant events I over which conditioning should be done. Note first that restricting the set of executions to the ones having a certain property amounts to conditioning on this property. In particular, we will condition on the fact that process i participates in round k. A crucial remark is that, in the worst case adversary framework that we are interested in, the adversary minimizing $\mathbf{P}_{\mathcal{A}}\left[W_i(k) \mid I\right]$ will make its choices as if "knowing" I. We will derive telling consequences from this fact.

We have actually in mind to compute the probability of $W_i(k)$ at different points s_k of the execution. One way to go, would be to condition on the past execution. But, by our previous remark, this is tantamount to allow the adversary to this knowledge. It is then easy to see that lockout is possible. Another natural alternative that we will adopt, is to compute the probability at point s_k "from the point of view of the adversary": this translates formally into conditioning over the value of the run up to point s_k. We will say that such a no-lockout property is *run-knowing*.

The first two definitions involve evaluating the probabilities "at the beginning of round k".

Definition 2.2 (Weak, Run-knowing, Probabilistic no-lockout) A solution to the mutual exclusion problem satisfies *weak, run-knowing* probabilistic no-lockout whenever there exists a constant c such that, for every fair adversary $\mathcal{A}$, every $k \geq 1$, every $(k-1)$-round run ρ compatible with $\mathcal{A}$, and every process i,

$$\mathbf{P}_{\mathcal{A}}\Big[W_i(k) \mid \pi_{k-1} = \rho,\ i \in \mathcal{P}(k)\Big] \geq c/n,$$

whenever $\mathbf{P}_{\mathcal{A}}[\pi_{k-1} = \rho,\ i \in \mathcal{P}(k)] \neq 0$.

The next property formally expresses statement (*) of Rabin. As we mentioned in our general presentation, considering rounds having m participating processes corresponds to conditioning on this fact.

Definition 2.3 (Strong, Run-knowing, Probabilistic no-lockout) The same as in Definition 2.2 except that:

$$\mathbf{P}_{\mathcal{A}}\Big[W_i(k) \mid \pi_{k-1} = \rho,\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m\Big] \geq c/m,$$

whenever $\mathbf{P}_{\mathcal{A}}[\pi_{k-1} = \rho,\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m] \neq 0$.

Recalling the two interpretations of conditioning in terms of time and knowledge held by the adversary, we see that this property differs fundamentally from the previous one because, here, the adversary is provided with the number of processes due to participate in the *future* round (i.e., after $t(k-1)$). By integration over m, we see that an algorithm satisfying the strong property also satisfies the weak property.

The next definition is the transcription of the previous one for the case where the probability is "computed at the beginning of the execution" (i.e., $s_k = 0$ for all k).

Definition 2.4 (Strong, Without knowledge, Probabilistic no-lockout) The same as in Definition 2.2 except that:

$$\mathbf{P}_{\mathcal{A}}\Big[W_i(k) \mid\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m\Big] \geq c/m,$$

whenever $\mathbf{P}_{\mathcal{A}}[i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m] \neq 0$.

By integration over ρ we see that an algorithm having the property of Definition 2.3 is stronger then one having the property of Definition 2.4. Equivalently, an adversary able to falsify Property 2.4 is stronger then one able to falsify Property 2.3.

[3] Recall that iid stands for "independent and identically distributed".

3 Our Results

Here, we give a little more detail about the operation of Rabin's algorithm than we gave earlier in the introduction. At each round k a new *round number* R is selected at random (uniformly among 100 values). The algorithm ensures that any process i that has already participated in the current round has $R_i = R$, and so passes a test that verifies this. The variable R acts as an "eraser" of the past: with high probability, a newly participating process does not pass this test and consequently chooses a new random number for its *lottery value* B_i. The distribution used for this purpose is a geometric distribution that is truncated at $b = \log_2 n + 4$: $\mathbf{P}\left[\beta_j(k) = l\right] = 2^{-l}$ for $l \leq b-1$. The first process that checks that its lottery value is the highest obtained so far in the round, at a point when the critical section is unoccupied, takes possession of the critical section. At this point the shared variable is reinitialized and a new round begins.

The algorithm has the following two features. First, any participating process i reinitializes its variable B_i at most once per round. Second, the process winning the competition takes at most two steps (and at least one) after the point f_k of the round at which the critical section becomes free. Equivalently, a process i that takes two steps after f_k and does not win the competition cannot hold the current maximal lottery value. (A process i having already taken a step in round k holds the current round number i.e., $R_i(k) = R(k)$. On the other hand, the semaphore S is set to 0 after f_k. If i held the highest lottery value it would pass all three tests in the code and enter the critical section.) We will take advantage of this last property in our constructions.

We are now ready to state our results. The first result states that the strong $\Omega(1/m)$ result claimed by Rabin is incorrect.

Theorem 3.1 The algorithm does not have the strong no-lockout property of Definition (2.4) (and hence of Definition 2.3). Indeed, there is an adversary $\mathcal{A}$ such that, for all rounds k, for all $m \leq n-1$, $\mathbf{P}_{\mathcal{A}}\left[1 \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m\right] \neq 0$ but $\mathbf{P}_{\mathcal{A}}\left[W_1(k) \mid 1 \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m\right] = 0$.

Proof: As we already remarked, the worst case adversary acts as if it knows the events on which conditioning is done. Knowing beforehand that the total number of participating processes in the round is m allows the adversary to design a schedule where processes take steps in turn, where process 1 begins and where process m takes possession of the critical section. Specifically, the adversary $\mathcal{A}$ does not use its knowledge about ρ, gives one step to process 1 while the critical section is occupied, waits for *Exit* and then adopts the schedule $2, 2, 3, 3, \ldots, n, n, 1$. This schedule brings round k to its end, because of the second property mentioned above (i.e., all processes are scheduled for two steps). For this adversary, for $2 \leq m \leq n-1$, $|\mathcal{P}(k)| = m$ happens exactly when process m wins so that $\mathbf{P}_{\mathcal{A}}[W_1(k) \cap |\mathcal{P}(k)| = m] = 0$. On the other hand, for this adversary, process m wins with non zero probability, i.e., $\mathbf{P}_{\mathcal{A}}[1 \in \mathcal{P}(k) \cap |\mathcal{P}(k)| = m] \neq 0$. ∎

```
Shared variable: V = (S, B, R), where:
        S ∈ {0,1}, initially 0
        B ∈ {0,1,...,⌈log n⌉ + 4}, initially 0
        R ∈ {0,2,...,99}, initially random

Code for i:
    Local variables:
        B_i ∈ {0,...,⌈log n⌉ + 4}, initially 1
        R_i ∈ {0,1,...,99}, initially ⊥
    Code:
    while V ≠ (0, B_i, R_i) do
        if (V.R ≠ R_i) or (V.B < B_i) then
            B_i ← random
            V.B ← max(V.B, B_i)
            R_i ← V.R
        unlock; lock;
    V ← (1, 0, random)
    unlock;
    * Critical Region **
    lock;
    V.S ← 0
    R_i ← ⊥
    B_i ← 0
    unlock;
    * Remainder Region **
    lock;
```

Figure 1: Rabin's Algorithm

The previous result is not too surprising in the light of the time interpretation given before Definition 2.2. restricting the execution to $\{|\mathcal{P}(k)| = m\}$ gives $\mathcal{A}$ too much knowledge about the *future*. We now give in Theorem 3.2 the more damaging result, stating (1) that, in spite of the randomization introduced in the round number variable R, the adversary is able to infer the values held in the local variables and (2) that it is able to use this knowledge to lock out a process with probability exponentially close to 1.

Theorem 3.2 There exists a constant $c < 1$, an adversary $\mathcal{A}$, a round k and a $k-1$-round run ρ such

that:

$$\mathbf{P}_{\mathcal{A}}[W_1(k) \mid \pi_{k-1} = \rho,\ 1 \in \mathcal{P}(k)] \leq e^{-32} + c^n.$$

We need the following definition in the proof.

Definition 3.1 Let l be a round. Assume that, during round l, the adversary adopts the following strategy. It first waits for the critical section to become free, then gives one step to process j and then *two* steps (in any order) to s other processes. (We will call these *test-processes.*) Assume that at this point the critical section is still available (so that round l is not over). We then say that process j is an s-survivor (at round l).

The idea behind this notion is that, by manufacturing survivors, the adversary is able to select processes having high lottery values. We now describe in more detail the selection of survivors and formalize this last fact.

In the following we will consider an adversary constructing sequentially a family of s-survivors for the four values $s = 2^{\log_2 n + t}$; $t = -1, \ldots, -5$. Whenever the adversary manages to select a new survivor it stores it, i.e, does not allocates it any further step until the selection of survivors is completed. ($\mathcal{A}$ actually allocates steps to selected survivors, but only very rarely, to comply with fairness. Rarely means for instance once every nT^2 steps, where T is the expected time to select an $n/2$-survivor.) By doing so, $\mathcal{A}$ reduces the pool of test-processes still available. We assume that, at any point in the selection process, the adversary selects the test-processes *at random* among the set of processes still available. (The adversary could be more sophisticated then random, but this is not needed.) Note that a new s-survivor can be constructed with probability one whenever the available pool has size at least $s+1$: it suffices to reiterate the selection process until the selection completes successfully.

Lemma 3.3 There is a constants d such that for any $t = -5, \ldots, -1$, for any $2^{\log_2 n + t}$-survivor j, for any $a = 0, \ldots, 5$

$$\mathbf{P}_{\mathcal{A}}[B_j(l) = \log n + t + a] \geq d.$$

Proof: Let s denote $\log n + t$. Let j be an s-survivor and $i_1, i_2, \ldots, i_s$ be the test-processes used in its selection. Assume also that j drew a new value $B_j(l) = \beta_j(l)$ (this happens with probability $q_1 = .99$.) Remark that $B_j(l) = \mathrm{Max}\{B_{i_1}(l), \ldots, B_{i_s}(l), B_j(l)\}$: if this were not the case, one of the test-processes would have entered *Crit*. As the test processes are selected at random, each of them has with probability .99 a round number different from $R(l)$ and hence draws a new lottery number $\beta_j(l)$. Hence, with high probability $q_2 > 0$, 90% of them do so. The other of them keep their old lottery value $B_j(l-1)$: this value, being old, has lost in previous rounds and is therefore stochastically smaller [4] then a new value $\beta_j(l)$. (An application of Lemma 4.5 formalizes this.) Hence, with probability at least $q_1 q_2$ we have the following stochastic inequality:

$$\mathrm{Max}\{\beta_1(l), \ldots, \beta_{s \cdot 90/100}\} \leq_{\mathcal{L}} B_j(l) \leq_{\mathcal{L}} \mathrm{Max}\{\beta_1(l), \ldots, \beta_{s+1}(l)\}.$$

Corollary 4.4 then shows that, for $a = 0, \ldots, 5$, with probability at least $q_1 q_2$, $\mathbf{P}_{\mathcal{A}}[B_j(l) = \log_2 s] \geq q_3$ for some constant q_3 (q_3 is close to 0.01). Hence, with probability at least $d \stackrel{\text{def}}{=} q_1 q_2 q_3$, $B_j(l)$ is equal to $\log_2 s + a$. ■

Proof of Theorem 3.2: The adversary uses a preparation phase to select and store some processes having high lottery values. We will, by abuse of language, identify this phase with the round ρ which corresponds to it. When this preparation phase is over, round k begins.

Preparation phase ρ: For each of the five values $\log_2 n + t$, $t = -5, \ldots, -1$, $\mathcal{A}$ selects in the preparation phase many ("many" means $n/20$ for $t = -5, \ldots, -2$ and $6n/20$ for $t = -1$) $2^{\log_2 n + t}$-survivors. Let S_1 denote the set of all the survivors thus selected. (Note that $|S_1| = n/2$ so that we have enough processes to conduct this selection). By partitioning the set of $2^{\log_2 n - 1}$-survivors into six sets of equal size, for each of the ten values $t = -5, \ldots, 4$, $\mathcal{A}$ has then secured the existence of $n/20$ processes whose lottery value is $\log_2 n + t$ with probability bigger then d. (By Lemma 3.3.)

Round k: While the critical section is busy, $\mathcal{A}$ gives a step to each of the $n/2$ processes from the set S_2 that it did not select in phase ρ. When this is done, with probability at least $1 - 2^{-32}$ (see Corollary 4.2) the program variable B holds a value bigger or equal then $\log_2 n - 5$. The adversary then waits for the critical section to become free and gives steps to the processes of S_1 it selected in phase ρ. A process in S_2 can win access to the critical section only if the maximum lottery value $B_{S_2} \stackrel{\text{def}}{=} \mathrm{Max}_{j \in S_2} B_j$ of all the processes in S_2 is strictly less then $\log_2 n - 5$ or if no process of S_1 holds both the correct round number $R(k)$ and the lottery number B_{S_2}. This consideration gives the bound predicted in Theorem 3.2 with $c = (1 - d/100)^{1/20}$. ■

Our proof actually demonstrates that there is an adversary that can lock out, with probability exponentially close to 1, an arbitrary set of $n/2$ processes

[4] A real random variable X is stochastically smaller then another one Y (we write that: $X \leq_{\mathcal{L}} Y$) exactly when, for all $x \in \mathbf{R}$, $\mathbf{P}[X \geq x] \leq \mathbf{P}[Y \geq x]$. Hence, if $X \leq Y$ in the usual sense, it is also stochastically smaller.

during *some* round. With a slight improvement we can derive an adversary that will succeed in locking out (with probability exponentially close to 1) a given set S_3 of, for example, $n/100$ processes at *all rounds*: we just need to remark that the adversary can do without this set S_3 during the preparation phase ρ. The adversary would then alternate preparation phases $\rho_1, \rho_2, \ldots$ with rounds $k_1, k_2, \ldots$ The set S_3 of processes would be given steps only during rounds $k_1, k_2, \ldots$ and would be locked out at each time with probability exponentially close to 1.

In view of our counterexample we might think that increasing the size of the shared variable might yield a solution. For instance, if the geometric distribution used by the algorithm is truncated at the value $b = 2\log_2 n$ instead of $\log_2 n + 4$, then the adversary is not able as before to ensure a lower bound on the probability that an $n/2$-survivor holds b as its lottery value. (The probability is given by Theorem 4.1 with $x = \log n$.) Then the argument of the previous proof does not hold anymore. Nevertheless, the next theorem establishes that raising the size of the shared variable does not help as long as the size stays sub-linear. But this is exactly the theoretical result the algorithm was supposed to achieve. (Recall the n-lower bound of [1] in the deterministic case.) Furthermore, the remark made above applies here also: a set of processes of linear size can be locked out at each time with probability arbitrarily close to 1.

Theorem 3.4 Suppose that we modify the algorithm so that the set of possible round numbers used has size r and that the set of possible lottery numbers has size b ($\log_2 n + 4 \leq b \leq n$). Then there exists positive constants c_1 and c_2, an adversary $\mathcal{A}$, and a run ρ such that

$$\mathbf{P}_{\mathcal{A}}[W_1(k) \mid \pi_{k-1} = \rho,\ 1 \in \mathcal{P}(k)] \leq e^{-32} + e^{-c_1 n/r} + c_2 \frac{r}{n^2}.$$

Proof: We consider the adversary $\mathcal{A}$ described in the proof of theorem 3.2: for $t = -5, \ldots, -2$, $\mathcal{A}$ prepares a set T_t of $2^{\log_2 n + t}$-survivors, each of size $n/20$, and a set T_{-1} of $2^{\log_2 n - 1}$-survivors; the size of T_{-1} is $6/20n$. (We can as before think of this set as being partitioned into six different sets.) We let η stand for $6/20$ in the sequel.

Let p_l denote the probability that process 1 holds l as its lottery value after having taken a step in round k. For any process j in S_{-1} let also q_l denote the probability that process j holds l as its lottery value at the end of the preparation phase ρ.

The same reasoning as in Theorem 3.2 then leads to the inequality:

$$\mathbf{P}_{\mathcal{A}}[W_1(k) \mid \pi_{k-1} = \rho,\ 1 \in \mathcal{P}(k)] \leq e^{-32} + (1 - e^{-32})(1 - d/r)^{n/20} + \sum_{l \geq \log_2 n + 5} p_l (1 - \frac{q_l}{r})^{\eta n}.$$

Write $l = \log_2 n + x - 1 = \log_2(n/2) + x$. Then, as is seen in the proof of Corollary 4.4, $q_l = e^{-2^{1-\zeta}} 2^{1-\zeta}$ for some $\zeta \in (x, x+1)$. For $l \geq \log_2 n + 5$, x is at least 6 and $e^{-2^{1-\zeta}} \sim 1$ so that $q_l \sim 2^{1-\zeta} \geq 2^{1-x}$. On the other hand $p_l = 2^{-l} = 2^{-x+1}/n$.

Define $\psi(x) \stackrel{\text{def}}{=} e^{-2^{1-x}\eta n/r}$ so that $\psi'(x) = e^{-2^{1-x}\eta n/r} 2^{1-x}\eta n/r$. Then:

$$\begin{aligned}
\sum_{l \geq \log_2 + 5} p_m (1 - \frac{q_m}{r})^{\eta n} &\leq 2/n \sum_{x \geq 6} 2^{-x} (1 - \frac{2^{1-x}}{r})^{\eta n} \\
&\leq 2/n \sum_{x \geq 6} 2^{-x} e^{-(\frac{2^{1-x}}{r}\eta n)} \\
&= 1/n \sum_{x \geq 6} 2^{1-x} e^{-(\frac{2^{1-x}}{r}\eta n)} \\
&= \frac{r}{\eta n^2} \sum_{x \geq 6} \psi'(x) \\
&\leq \frac{r}{\eta n^2} \int_5^\infty \psi'(x) dx \\
&= \frac{r}{\eta n^2} [\psi]_5^\infty \\
&= \frac{r}{\eta n^2} [1 - e^{-2^{-4}\eta n/r}] \\
&\leq \frac{r}{\eta n^2}.
\end{aligned}$$

■

To simplify the notations in the sequel, we will let $i_1, \ldots, i_{|\mathcal{P}(k)|}$ denote the elements of $\mathcal{P}(k)$. And we will let $p_1, p_2, \ldots$ denote the sequence of processes taking steps in turn during round k: recall that a process i can take several steps during the round.

The flaw of the protocol revealed in Theorem 3.2 is based on the fact that the variable R does not act as an eraser of the past and that the adversary can use old values to defeat the algorithm. The flaw exhibited in Theorem 3.1 is based on the fact that, even when the old values are erased, the algorithm is sensitive to the order $p_1, p_2, \ldots$ in which participating processes are scheduled. The adversary can play on this order in two different ways. It can act on the fact that different scheduling strategies influence in different ways the size m of the set $\mathcal{P}(k)$ (Strategy 1). And it can use the fact that, for a *given* number m of participating processes, the mathematical distribution of the sequence $(\beta_i(k);\ i \in \mathcal{P}(k))$ is (a priori) sensitive to the ordering $p_1, p_2, \ldots$ (Strategy 2). The adversary of Theorem 3.1 specifically used strategy 1.

The next result shows that the two flaws exhibited in Theorems 3.1 and 3.2 are at the core of the prob-

lem: the algorithm does have the strong no-lockout property when we precondition on the fact that the internal variables of the participating processes are reset to new values and when we bar the adversary from using strategy 1. We will actually prove this result for a slightly modified version of the algorithm. Recall in effect that the code given in Page 6 is optimized by making a participating process i draw a new lottery number when it is detected that $V.B < B_i$. We will consider the "de-optimized" version of the code in which only the test $V.R \neq R_i$? causes of a new drawing to occur.

The next definition formalizes the restriction that we impose on the adversary. It says that the adversary commits itself to the value of $\mathcal{P}(k)$ at the beginning of round k.

Definition 3.2 We say that an adversary is *restricted* when, for each round, it allocates a step to all participating processes (of this round) before the critical section becomes free. We will let $\mathcal{A}'$ (as opposed to $\mathcal{A}$) denote any such adversary.

We will make constant use of the notation $[n] \stackrel{\text{def}}{=} \{1, 2, \ldots, n\}$. Also, for any sequence $(a_j)_{j \in \mathbf{N}}$ we will write $a_i = \underset{j \in J}{\text{Umax}} a_j$ to mean that i is the *only* index in J for which $a_i = \underset{j \in J}{\text{Max}} a_j$.

Theorem 3.5 For every process $i = 1, \ldots, n$, for every round $k \geq 1$, for every restricted adversary $\mathcal{A}'$ and for every $(k-1)$-round run ρ compatible with $\mathcal{A}'$,
$\mathbf{P}_{\mathcal{A}'}[W_i(k) \mid \mathcal{N}(k),\ \pi_{k-1} = \rho,\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m] \geq \frac{2}{3m}$, whenever
$\mathbf{P}_{\mathcal{A}'}[\mathcal{N}(k),\ \pi_{k-1} = \rho,\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m] \neq 0$.

Proof:

We first define the events $\mathcal{U}(k)$ and $\mathcal{U}'_J(k)$, where J is any subset of $\{1, \ldots, n\}$:

$$\mathcal{U}(k) \stackrel{\text{def}}{=} \{\exists! i \in \mathcal{P}(k) \text{ s.t. } B_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Max}} B_j(k)\},$$

$$\mathcal{U}'_J(k) \stackrel{\text{def}}{=} \{\exists! i \in J \text{ s.t. } \beta_i(k) = \underset{j \in J}{\text{Max}} \beta_j(k)\}.$$

The main result established in [6] can formally be restated as:

$$\forall m \leq n,\ \mathbf{P}\Big[\mathcal{U}'_{[m]}(k)\Big] \geq 2/3. \qquad (1)$$

Following the general proof technique described in the introduction we will prove that :

$$\mathbf{P}_{\mathcal{A}'}\Big[\mathcal{U}(k) \mid \mathcal{N}(k),\ \pi_{k-1} = \rho,\ i \in \mathcal{P}(k), |\mathcal{P}(k)| = m\Big] = \mathbf{P}\Big[\mathcal{U}'_m(k)\Big]\ ,\ \text{and that:}$$

$$\mathbf{P}_{\mathcal{A}'}\Big[W_i(k) \mid \mathcal{N}(k), \pi_{k-1} = \rho, i \in \mathcal{P}(k), |\mathcal{P}(k)| = m, \mathcal{U}(k)\Big] = \mathbf{P}\Big[\beta_i(k) = \underset{j \in [m]}{\text{Max}} \beta_j(k) \mid \mathcal{U}'_m(k)\Big]\ .$$

The events involved in the LHS of the two inequalities (e.g., $W_i(k)$, $\mathcal{U}(k)$, $\{|\mathcal{P}(k)| = m\}$, $\{\pi_{k-1} = \rho\}$, $\{i \in \mathcal{P}(k)\}$) depend on $\mathcal{A}'$ whereas the events involved in the RHS are pure mathematical events over which $\mathcal{A}'$ has no control.

We begin with some important remarks.

(1) By definition, the set $\mathcal{P}(k) = \{i_1, i_2, \ldots\}$ is decided by the restricted adversary $\mathcal{A}'$ at the beginning of round k: for a given $\mathcal{A}'$ and conditioned on $\{\pi_{k-1} = \rho\}$, the set $\mathcal{P}(k)$ is defined *deterministically*. In particular, for any i, $\mathbf{P}_{\mathcal{A}'}[\, i \in \mathcal{P}(k) \mid \pi_{k-1} = \rho]$ has value 0 or 1. Similarly, there is one value m for which $\mathbf{P}_{\mathcal{A}'}[|\mathcal{P}(k)| = m \mid \pi_{k-1} = \rho] = 1$. Hence, for a given adversary $\mathcal{A}'$, if the random event $\{\mathcal{N}(k),\ \pi_{k-1} = \rho,\ i \in \mathcal{P}(k),\ |\mathcal{P}(k)| = m\}$ has non zero probability, it is equal to the random event $\{\mathcal{N}(k),\ \pi_{k-1} = \rho\} \stackrel{\text{def}}{=} I$.

(2) Recall that, in the modified version of the algorithm that we consider here, a process i draws a new lottery value in round k exactly when $R_i(k-1) \neq R(k)$. Hence, within I, the event $\mathcal{N}(k)$ is equal to $\{R_{i_1}(k-1) \neq R(k), \ldots, R_{i_m}(k-1) \neq R(k)\}$. On the other hand, by definition, the random variables (in short r.v.s) β_{i_j}; $i_j \in \mathcal{P}(k)$ are iid and independent from the r.v. $R(k)$. This proves that, (for a given $\mathcal{A}'$), conditioned on $\{\pi_{k-1} = \rho\}$, the r.v. $\mathcal{N}(k)$ is independent from all the r.v.s β_{i_j}. Note that $\mathcal{U}'_{\mathcal{P}(k)}(k)$ is defined in terms of (i.e., measurable with respect to) the $(\beta_{i_j};\ i_j \in \mathcal{P}(k))$, so that $\mathcal{U}'_{\mathcal{P}(k)}(k)$ and $\mathcal{N}(k)$ are also independent.

(3) More generally, consider any r.v. X defined in terms of the $(\beta_{i_j};\ i_j \in \mathcal{P}(k))$: $X = f(\beta_{i_1}, \ldots, \beta_{i_m})$ for some measurable function f. Recall once more that the number m and the *indices* $i_1, \ldots, i_m$ are determined by $\{\pi_{k-1} = \rho\}$ and $\mathcal{A}'$. The r.v.s β_{i_j} being iid, for a fixed $\mathcal{A}'$, X then depends on $\{\pi_{k-1} = \rho\}$ only through the value m of $|\mathcal{P}(k)|$. Formally, this means that, conditioned on $|\mathcal{P}(k)|$, the r.v.s X and $\{\pi_{k-1} = \rho\}$ are independent: $\mathbf{E}_{\mathcal{A}'}[X \mid \pi_{k-1} = \rho\,] = \mathbf{E}_{\mathcal{A}'}[X \mid |\mathcal{P}(k)| = m] = \mathbf{E}[f(\beta_1, \ldots, \beta_m)]$. (More precisely, this equality is valid for the value m for which $\mathbf{P}_{\mathcal{A}}[\pi_{k-1} = \rho\ ,\ |\mathcal{P}(k)| = m] \neq 0$.) A special consequence of this fact is that $\mathbf{P}_{\mathcal{A}'}[\mathcal{U}'_{\mathcal{P}(k)}(k) \mid \pi_{k-1} = \rho] = \mathbf{P}[\mathcal{U}'_{[m]}(k)]$.

Remark that, in $\mathcal{U}(k)$, the event $W_i(k)$ is the same as the event $\{B_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} B_j(k)\}$. This justifies the first following equality. The subsequent ones are commented afterwards. Also, the set I that we consider here is the one having a non zero probability described in Remark (1) above.

$$\begin{aligned}
&\mathbf{P}_{\mathcal{A}'}[W_i(k) \mid \mathcal{U}(k),\ I] \\
&= \mathbf{P}_{\mathcal{A}'}[B_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} B_j(k) \mid \mathcal{U}(k),\ I\ ] \\
&= \mathbf{P}_{\mathcal{A}'}[\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} \beta_j(k) \mid \mathcal{U}'_{\mathcal{P}(k)}(k),\ I\ ] \qquad (2) \\
&= \mathbf{P}_{\mathcal{A}'}[\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} \beta_j(k) \mid \mathcal{U}'_{\mathcal{P}(k)}(k), \pi_{k-1} = \rho] \qquad (3)
\end{aligned}$$

Equation 2 is true because we condition on $\mathcal{N}(k)$ and because $\mathcal{U}(k) \cap \mathcal{N}(k) = \mathcal{U}'_{\mathcal{P}(k)}(k)$. Equation 3 is true because $\mathcal{N}(k)$ is independent from the r.v.s β_{i_j} as is shown in Remark (2) above.

We then notice that the events $\{\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} \beta_j(k)\}$ and $\mathcal{U}'_{\mathcal{P}(k)}(k)$ (and hence their intersection) are defined in terms of the r.v.s β_{i_j}. From remark (3) above, the value of Eq. 3 depends only on m and is therefore independent of i. Hence, for all i and j in $\mathcal{P}(k)$, $\mathbf{P}_{\mathcal{A}'}[W_i(k) \mid \mathcal{U}(k), I] = \mathbf{P}_{\mathcal{A}'}[W_j(k) \mid \mathcal{U}(k), I]$.

On the other hand, $\sum_{i \in \mathcal{P}(k)} \mathbf{P}_{\mathcal{A}'}[\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Umax}} \beta_j(k) \mid \mathcal{U}'_{\mathcal{P}(k)}(k), \pi_{k-1} = \rho] = 1$: indeed, one of the β_{i_j} has to attain the maximum.

These last two facts imply that, $\forall i \in \mathcal{P}(k)$,

$$\mathbf{P}_{\mathcal{A}'}[W_i(k) \mid \mathcal{U}(k), I] = 1/m.$$

We now turn to the evaluation of $\mathbf{P}_{\mathcal{A}'}[\mathcal{U}(k) \mid I]$.

$$\begin{aligned} \mathbf{P}_{\mathcal{A}'}[\mathcal{U}(k) \mid I] &= \mathbf{P}_{\mathcal{A}'}[\mathcal{U}'_{\mathcal{P}(k)}(k) \mid I] && (4) \\ &= \mathbf{P}_{\mathcal{A}'}[\mathcal{U}'_{\mathcal{P}(k)}(k) \mid \pi_{k-1} = \rho] && (5) \\ &= \mathbf{P}[\mathcal{U}'_{[m]}(k)] \geq 2/3 . && (6) \end{aligned}$$

Equation 4 is true because we condition on $\mathcal{N}(k)$. Eq. 5 is true because $\mathcal{U}'_{\mathcal{P}(k)}(k)$ and $\mathcal{N}(k)$ are independent (See Remark (2) above). The equality of Eq. 6 stems from Remark (3) above and the inequality from Eq. 1.

We can now finish the proof of Theorem 3.5.

$$\begin{aligned} &\mathbf{P}_{\mathcal{A}'}[W_i(k) \mid I] \\ &\quad \geq \mathbf{P}_{\mathcal{A}'}[W_i(k), \mathcal{U}(k) \mid I] \\ &\quad = \mathbf{P}_{\mathcal{A}'}[W_i(k) \mid \mathcal{U}(k), I] \, \mathbf{P}_{\mathcal{A}'}[\mathcal{U}(k) \mid I] \geq 2/3 \, m . \end{aligned}$$

■

We discuss here the lessons brought by our results. (1) Conditioning on $\mathcal{N}(k)$ is equivalent to force the algorithm to refresh all the variables at each round. By doing this, we took care of the undesirable lingering effects of the past, exemplified in Theorems 3.2 and 3.4. (2) It is *not* true that:

$$\mathbf{P}_{\mathcal{A}}\Big[\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Max}} \beta_j(k) \mid \mathcal{U}'_{\mathcal{P}(k)}(k), |\mathcal{P}(k)| = m\Big] = \mathbf{P}\Big[\beta_i(k) = \underset{j \in [m]}{\text{Max}} \beta_j(k) \mid \mathcal{U}'_{[m]}(k)\Big],$$

i.e., that the adversary has no control over the event $\{\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Max}} \beta_j(k)\}$. (This was Rabin's statement in [6].)

Indeed, the latter probability is equal to $1/m$ whereas we proved in Theorem 3.1 that there is an adversary for which the former is 0 when $m \leq n-1$.

The crucial remark explaining this apparent paradox is that, implicit in the expression $\mathbf{P}_{\mathcal{A}}[\beta_i(k) = \underset{j \in \mathcal{P}(k)}{\text{Max}} \beta_j(k) \mid \ldots]$, is the fact that the random variables $\beta_j(k)$ (for $j \in \mathcal{P}(k)$) are compared to each other in a specific way decided by $\mathcal{A}$, before one of them reveals itself to be the maximum. For instance, in the example constructed in the proof of Theorem 3.1, when j takes a step, $\beta_j(k)$ is compared *only* to the $\beta_l(k)$; $l \leq j$, and the situation is not symmetric among the processes in $\mathcal{P}(k)$.

But, if the adversary is restricted as in our Definition 3.2, the symmetry is restored and the strong no-lockout property holds.

Rabin and Kushilevitz used these ideas from our analysis to produce their algorithm [7].

In our last Theorem 3.5 we used the restriction on the adversary $\mathcal{A}'$ mostly to derive a $1/m$ bound. If we consider a general adversary $\mathcal{A}$ it is interesting to note that we can still ensure the weak lockout-property:

Theorem 3.6 For every process $i = 1, \ldots, n$, for every round $k \geq 1$, for every adversary $\mathcal{A}$ and for every $(k-1)$-round run ρ compatible with $\mathcal{A}$,
$\mathbf{P}_{\mathcal{A}}\Big[W_i(k) \mid \mathcal{N}(k), \pi_{k-1} = \rho, i \in \mathcal{P}(k)\Big] \geq .1/n$,
whenever $\mathbf{P}_{\mathcal{A}}\Big[\mathcal{N}(k), \pi_{k-1} = \rho, i \in \mathcal{P}(k)\Big] \neq 0$.

Proof: Omitted. ■

This theorem holds also if, as in the context of theorem 3.4, the algorithm uses b lottery numbers. This shows that the result of Theorem 3.6 is not trivial: indeed, when $b = 2\log 2$, the probability $\mathbf{P}[\beta_i(k) = b]$ of drawing the highest possible number is a $o(1/n)$. One of the difficulties of the proof is that the apparently innocuous event $\{i \in \mathcal{P}(k)\}$ is in the future of the point $t(k-1)$ at which the probability is estimated: the adversary could conceivably also use this fact to ensure some specific values of the variables when i participates.

Our Theorems 3.1, 3.2 and 3.4 explored how the adversary can gain and use knowledge of the lottery values held by the processes. The next theorem states that the adversary is similarly able to derive some knowledge about the round numbers, contradicting the claim in [6] that "because the variable R is randomized just before the start of the round, we have with probability 0.99 that $R_i \neq R$." Note that, expressed in our terms, the previous claim translates into $R(k) \neq R_i(k-1)$.

Theorem 3.7 There exists an adversary $\mathcal{A}$, a round k, a step number t, a run ρ_t, compatible with $\mathcal{A}$, having t steps and in which round k is under way such that

$$\mathbf{P}_{\mathcal{A}}[R(k) \neq R_1(k-1) \mid \pi_t = \rho_t] < .99 .$$

Proof:

We will write $\rho_t = \rho'\rho$ where ρ' is a $k-1$-round run and ρ is the run fragment corresponding to the kth round under way. Assume that ρ' indicates that, before round k, processes $1,2,3,4$ participated *only* in round $k-1$, and that process 5 never participated before round k. Furthermore, assume that during round $k-1$ the following pattern happened: $\mathcal{A}$ waited for the critical region to become free, then allocated one step in turn to processes $2,1,1,3,3,4,4$; at this point 4 entered the critical region. (All this is indicated in ρ'.) Assume also that the partial run ρ into round k indicates that the critical region became free before any competing process was given a step, and that the adversary then allocated one step in turn to processes $5,3,3$, and that, after 3 took its last step, the critical section was still free. We will establish that, at this point,

$$\mathbf{P}_{\mathcal{A}}[R(k) \neq R_1(k-1) \mid \pi_t = \rho'\rho] < .99\ .$$

By assumption $k-1$ is the last (and only) round before round k where processes $1,2,3$ and 4 participated. Hence $R_1(k-1) = R_2(k-1) = R_3(k-1) = R(k-1)$. To simplify the notations we will let R' denote this common value. Similarly we will write β_1', β_2', ... in place of $\beta_1(k-1)$, $\beta_2(k-1)$, ... We will furthermore write β_1, β_2, ... in place of $\beta_1(k)$, $\beta_2(k)$, ... and B, R in place of $B(k)$, $R(k)$.

Using Bayes' rule gives us:

$$\begin{aligned}&\mathbf{P}_{\mathcal{A}}[R \neq R' \mid \rho',\ \rho]\\ &= \frac{\mathbf{P}_{\mathcal{A}}[R \neq R' \mid \rho']\ \mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R \neq R']}{\mathbf{P}_{\mathcal{A}}[\rho \mid \rho']}. \qquad (7)\end{aligned}$$

In the numerator, the first term $\mathbf{P}_{\mathcal{A}}[R \neq R' \mid \rho']$ is equal to 0.99 because R is uniformly distributed and independent from R' and ρ'. We will use this fact another time while expressing the value of $\mathbf{P}_{\mathcal{A}}[\rho \mid \rho']$:

$$\begin{aligned}&\mathbf{P}_{\mathcal{A}}[\rho \mid \rho']\\ &= \mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R \neq R']\ \mathbf{P}_{\mathcal{A}}[R \neq R' \mid \rho']\\ &\quad + \mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R = R']\ \mathbf{P}_{\mathcal{A}}[R = R' \mid \rho']\\ &= 0.99\ \mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R \neq R'] \qquad (8)\\ &\quad + 0.01\ \mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R = R'].\end{aligned}$$

• Consider first the case where $R \neq R'$. Then process 3 gets a YES answer when going through the test "$(V.R \neq R_3)$ **or** $(V.B < B_3)$", and consequently chooses a new value $B_3(k) = \beta_3$. Hence

$$\mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R \neq R'] = \mathbf{P}[\beta_3 < \beta_5]. \qquad (9)$$

• Consider now the case $R = R'$. By hypothesis, process 5 never participated in the computation before round k and hence draws a new number $B_5(k) = \beta_5$. Hence:

$$\begin{aligned}&\mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R = R'] =\\ &\quad \mathbf{P}_{\mathcal{A}}[B_3(k) < \beta_5 \mid \rho', R = R']. \qquad (10)\end{aligned}$$

As processes $1,\ldots,4$ participated *only* in round $k-1$ up to round k, the knowledge provided by ρ' about process 3 is exactly that, in round $k-1$, process 3 lost to process 2 along with process 1, and that process 2 lost in turn to process 4, i.e., that $\beta_3' < \beta_2'$, $\beta_1' < \beta_2'$ and $\beta_2' < \beta_4'$. For the sake of notational simplicity, for the rest of this paragraph we let X denote a random variable whose law is the law of β_2' conditioned on $\{\beta_2' > \mathrm{Max}\{\beta_1', \beta_3'\}, \beta_2' < \beta_4'\}$. This means for instance that, $\forall x \in \mathbb{R}$,

$$\mathbf{P}[X \geq x] = \mathbf{P}\Big[\beta_2' \geq x \mid \beta_2' > \mathrm{Max}\{\beta_1', \beta_3'\},\ \beta_2' < \beta_4'\Big].$$

When 3 takes its first step within round k, the program variable $V.B$ holds the value β_5. As a consequence, 3 chooses a new value when and exactly when $B_3(k-1)(= \beta_3')$ is strictly bigger then β_5. (The case $\beta_3' = \beta_5$ would lead 3 to take possession of the critical section at its first step in round k, in contradiction with the definition of ρ; and the case $\beta_3' < \beta_5$ leads 3 to keep its "old" lottery value $B_3(k-1)$.) From this we deduce that:

$$\begin{aligned}&\beta_3(k) < \beta_5 \mid \rho', R = R'] = \mathbf{P}[\beta_3' < \beta_5 \mid \beta_3' < X\\ &\quad + \mathbf{P}[\beta_3' > \beta_5,\ \beta_3 < \beta_5 \mid \beta_3' < X]. \qquad (11)\end{aligned}$$

Using Lemma 4.5 we derive that:

$$\mathbf{P}[\beta_3' < \beta_5 \mid \beta_3' < X] \geq \mathbf{P}[\beta_3' < \beta_5].$$

On the other hand $\mathbf{P}[\beta_3' < \beta_5] = \mathbf{P}[\beta_3 < \beta_5]$ because all the random variables $\beta_i(j), i = 1,\ldots,n,\ j \geq 1$ are iid. Taking into account the fact that the last term of equation 11 is non zero, we have then established that:

$$\mathbf{P}_{\mathcal{A}}[B_3(k) < \beta_5 \mid \rho', R = R'] > \mathbf{P}[\beta_3 < \beta_5] \qquad (12)$$

Combining Equations 9, 10 and 12 yields:

$$\mathbf{P}_{\mathcal{A}}[\rho \mid \rho', R = R'] > \mathbf{P}_{\mathcal{A}}[\rho \mid \rho',\ R \neq R'].$$

Equation 8 then shows that $\mathbf{P}_{\mathcal{A}}[\rho \mid \rho'] > \mathbf{P}_{\mathcal{A}}[\rho \mid \rho',\ R \neq R']$. Plugging this result into Equation 7 finishes the proof. ■

We finish with a result showing that all the problems that we encountered in Rabin's algorithm carry over for Ben-Or's algorithm. Ben-Or's algorithm is cited at the end of [6]. The code of this algorithm is the same as the one of Rabin with the following modifications. All variables $B, R, B_i, R_i; 1 \leq i \leq n$ are

boolean variables, initially 0. The distribution of the lottery numbers is also different but this is irrelevant for our discussion.

We show that Ben-Or's algorithm does not satisfy the weak no-lockout property of Definition 2.2. The situation is much simpler then in the case of Rabin's algorithm: here all the variables are boolean so that a simple reasoning can be worked out.

Theorem 3.8 (Ben Or's Alg.) There is an adversary $\mathcal{A}$, a step number t and a run ρ_t compatible with $\mathcal{A}$ such that

$$\mathbf{P}_{\mathcal{A}}\Big[W_2(k) \mid \pi_t = \rho_t,\ 2 \in \mathcal{P}(k)\Big] = 0\ .$$

Proof: Assume that we are in the middle of round 3, and that the run ρ_t indicates that (at time 0 the critical section was free and then that) the schedule 1 2 2 3 3 was followed, that at this point 3 entered in *Crit*, that it left *Crit*, that at this point the schedule 4 1 1 5 5 was followed, that 5 entered and then left *Crit*, that 6 4 4 then took a step and that at this point *Crit* is still free.

Without loss of generality assume that the round number $R(1)$ is 0. Then $R_2(1) = 0$, $B_1(1) = 1$ and $B_2(1) = 0$: if not 2 would have entered in *Crit*. In round 2 it then must be the case that $R(2) = 1$. Indeed if this was not the case then 1 would have entered the critical section. It must then be the case that $B_1(2) = 0$ and $B_4(2) = 1$. And then that $B_6(3) = 1$ and $R(3) = 0$: if this was not the case then 4 would have entered in *Crit* in the 3rd round.

But at this point, 2 has *no chance* to win if scheduled to take a step! ∎

Acknowledgments I am deeply indebted to Nancy Lynch who suggested the problem and who constantly assisted me: this paper is hers too.

References

[1] Burns J., Fischer M., Jackson P., Lynch N. and Peterson G. Data requirements for implementation of n- process mutual exclusion using a single shared variable. *Journal of the ACM*, 29:183-205, (1982).

[2] E. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 321, (1966).

[3] Flajolet P. and Martin N. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31:182–209, (1985).

[4] Graham R. and Yao A. On the Improbability of Reaching Byzantine Agreements *Proc. 21st ACM Symp. on Theory of Computer Science* 467–478 (1989).

[5] Hart S., Sharir M. and Pnueli A. Termination of Probabilistic Concurrent Programs *ACM Transactions on Programming Languages and Systems*, Vol 5, Num 3:356–380, (1983).

[6] Michael Rabin. N-process mutual exclusion with bounded waiting by 4 log N- shared variable. *Journal of Computation and System Sciences*, 25:66–75 (1982).

[7] Rabin M. and Kushilevitz E. Randomized Mutual Exclusion Algorithm Revisited *This proceedings*

[8] Saias I. and Lynch N. An Analysis of Rabin's Randomized Mutual Exclusion Algorithm. *MIT/LCS/TM-462* (1991).

4 Appendix

Theorem 4.1 and its corollaries are used in the construction of the adversary in Theorem 3.2 and Theorem 3.4. Lemma 4.5 is used mostly in the proof of Theorem 3.7. The proofs can be found in [8].

Definition 4.1 For any sequence $(a_i)_{i\in\mathbf{N}}$ we denote $\mathsf{Max}_s a_i \stackrel{\text{def}}{=} \mathsf{Max}\{a_1, a_2, \ldots, a_s\}$.

In this section the sequence (β_i) is a sequence of iid geometric random variables:

$$\mathbf{P}[\beta_i = l] = \frac{1}{2^l};\ l = 1, 2, \ldots$$

The following results are about the distribution of the extremal function $\mathsf{Max}_s\beta_i$. The same probabilistic results hold for iid random variables (β'_i), having the truncated distribution used by Rabin: we just need to truncate at $\log_2 n + 4$ the random variables β_i and the values that they take. This does not affect the probabilities because, by definition, $\mathbf{P}[\beta'_i(k) = \log_2 n + 4] = \sum_{l \geq \log_2 n + 4} \mathbf{P}[\beta_i = l]$.

Theorem 4.1 For $\frac{2}{s}^{1-x} \leq 1/2$ we have the following approximation:

$$A \stackrel{\text{def}}{=} \mathbf{P}[\mathsf{Max}_s\beta_i \geq \log_2 s + x] \sim 1 - e^{-2^{1-x}}\ .$$

$$\left|A - e^{-2^{1-x}}\right| \leq e^{-2^{1-x}} \frac{4^{1-x}}{s}\ .$$

Corollary 4.2 $\mathbf{P}[\mathsf{Max}_s\beta_i \geq \log_2 s - 4] \geq 1 - e^{-32}$.

Corollary 4.3 $\mathbf{P}[\mathsf{Max}_s\beta_i \geq \log_2 s + 8] \leq 0.01$.

Corollary 4.4 $\mathbf{P}[\mathsf{Max}_s\beta_i = \log_2 s] \geq 0.17$,
$\mathbf{P}[\mathsf{Max}_s\beta_i = \log_2 s + l] \geq 0.01,\ \forall l = 1, \ldots, 5$.

Lemma 4.5 Let B and A be any real-valued random variables. Then

$$\forall x \in \mathbb{R},\ \mathbf{P}[B \geq x \mid B \leq A] \leq \mathbf{P}[B \geq x].$$ [5]

[5] We use the convention that $0/0 = 0$ whenever this quantity arises in the computation of conditional probabilities.

Randomized Mutual Exclusion Algorithms Revisited*

Eyal Kushilevitz† Michael O. Rabin‡

Abstract

In [4] a randomized algorithm for mutual exclusion with bounded waiting, employing a logarithmic sized shared variable, was given. Saias and Lynch [5] pointed out that the adversary scheduler postulated in the above paper can observe the behavior of processes in the interval between an opening of the critical section and the next closing of the critical section. It can then draw conclusions about values of their local variables as well as the value of the randomized round number component of the shared variable, and arrange the schedule so as to discriminate against a chosen process. This invalidates the claimed properties of the algorithm.

In the present paper the algorithm in [4] is modified, using the ideas of [4], so as to overcome this difficulty, obtaining essentially the same results. Thus, as in [4], randomization yields simple algorithms for mutual-exclusion with bounded waiting, employing a shared variable of considerably smaller size than the lower-bound established in [1] for deterministic algorithms.

*Research supported by research contracts ONR-N0001491-J-1981 and NSF-CCR-90-07677.

†Aiken Computation Lab., Harvard University and Computer Science Dept., Technion. e-mail: eyalk@das.harvard.edu .

‡Aiken Computation Lab., Harvard University and Institute of Mathematics, Hebrew University of Jerusalem. e-mail: rabin@das.harvard.edu .

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0275...$1.50

1 Introduction

In this paper we deal with the well-known *mutual-exclusion* problem: Let $P_1, \cdots, P_N$ be N processes that from time to time need to execute a critical section in which exactly one is allowed to employ some shared resource. They can coordinate their activities by use of a shared *test-and-set* variable v (i.e., testing and setting v itself is an atomic action, and access to v is always available to a P_i scheduled to do so). This problem was suggested by Dijkstra [2] and was discussed in many papers since then (see, for example, [1, 4] and the literature cited there). A solution for this problem is an algorithm that guarantees freedom from *deadlock* (this alone can be achieved by the use of a one-bit semaphore) and freedom from *lockout*.

Burns et. al. [1] considered the following question: What should be the size of the shared variable v so that (deadlock-free, lockout-free) mutual-exclusion can be implemented? This question is not only of theoretical interest but also of practical interest. This is because in practice test-and-set is not an atomic operation and what we really assume is that reading the variable and immediately writing it can be done very fast so that

no other operations will interrupt it. Such an assumption is reasonable only for "short" variables. Burns et. al. [1] proved that if *deterministic* algorithms are used then an $\Omega(\log N)$-bit shared variable is required, and in fact this number of bits is also sufficient.

Rabin [4], presented a *randomized* solution for the problem using an $O(\log \log N)$-bit shared variable. His algorithm was based on the following lemma:

> For *any* $1 \leq m \leq N$ processes, say $P_1, \cdots, P_m$, if each P_i randomly, with a *geometric* distribution, draws a number $1 \leq b^{(i)} \leq \log N + 4$, then with probability at least 2/3 $b^{(j)} = \max b^{(i)}$ will hold for exactly one j.

The algorithm works in rounds, where a round is defined to be the time between two successive entrances to the critical section. To explain the algorithm we provisionally assume that the shared variable v contains a field, updated by the process entering the critical section, representing the current round number r. In addition the shared variable contains a flag to indicate whether the critical section is close or open, and a field b that contains the maximum number drawn during this round. Each process trying to enter the critical section during round r, upon accessing the shared variable, draws a number according to the geometric distribution, and updates the field b in the shared variable by the maximum among his number and the current value of b. If it already drew a number in round r it does *not* draw a number again. If the critical section is open and the number it drew equals b (the maximal number drawn in this round), it enters the critical section and starts a new round.

This solution is not only deadlock-free and lockout-free but also satisfies (using the above lemma) a powerful fairness property: If a process P_i participates in a trying round together with m other processes, it has a probability of $\Omega(1/m)$ to enter the critical section at the end of this trying round. This property is called in [4] *bounded waiting*. It was also shown, based on an idea of Ben-Or, that this algorithm can be modified so as to use just a constant size shared variable. This version, however, guarantees only a weaker fairness property: If a process P_i participates in a trying round, it has a probability of $\Omega(1/N)$ to enter the critical section at the end of this trying round. This is a much weaker property since in practice m, the number of processes competing for the critical section during a trying round, is typically much smaller than N, the number of processes in the system. It is important to remark that in both versions of the algorithm deadlock is never possible.

The difficulty with these algorithms is that we assumed that the unbounded round number r is part of the shared variable v. (All the other fields of v are of the appropriate size.) The idea for dealing with this problem was to replace the use of the round number r by a *randomized round number* (i.e., a random bit chosen by the process entering the critical section). The intuition was that a randomized round number is enough to guarantee that a process will not draw a number more than once at a trying round, and it seemed that the probability that a process will not draw a number at all is exactly 1/2. Therefore, the same analysis seemed to work.

Recently, Saias and Lynch [5] showed that this is not true. They presented some examples in which an adversary scheduler can lockout a process P_i. Summarizing these examples there are two problems with the use of randomized round number instead of the actual round number in the above algorithms:

- Processes that lost in one trying round (i.e., did not enter the critical section) may remain with "high" lottery numbers. Such an occurance can be observed by the adversary according to the external behavior of the processes (without looking at the content of their local variables nor at the shared variable). Later, these processes can participate in a trying round which has the same randomized round number, together with the process P_i. Therefore, they will not draw new numbers (and remain with the old "high" numbers) and hence the probability of P_i to win the lottery in such a case is smaller than it should be.

- The adversary can learn whether the current randomized round number equals the randomized round number in the last trying round P_i participated in, by observing the external behavior of other processes participated with P_i in that previous round. Then, the adversary can schedule P_i only in rounds with same randomized round number. This will cause P_i not to draw a new number and this again decreases the probability of P_i to win the lottery.

We modify the algorithms presented in [4] using shared variables with the same number of bits (up to a constant) achieving the same fairness properties. The key ideas for overcoming the above two problems are:

- The modified algorithms make sure that processes which lost in a lottery will not keep high lottery numbers that can be used in future rounds.

- In the modified algorithms, each trying round is divided into two parts: the *drawing round* in which the processes draw their numbers, and the *notification round* in which the processes that took part in the drawing round find out who win and who lost. The idea is that during the drawing round, the external behavior of processes does not depend on the outcome of the lottery, and therefore during this time the adversary cannot gain information about the contents of the shared variable nor the local variables of the processes. During the notification round, the lottery is already over and the winner and the losers are already determined.

We believe that the separation idea may be useful in the design of other randomized protocols. To summarize: randomization yields simple algorithms for mutual-exclusion with bounded waiting, employing a shared variable of considerably smaller size than the lower-bound for deterministic algorithms.

2 The Modified Algorithm

2.1 Definitions and General Plan

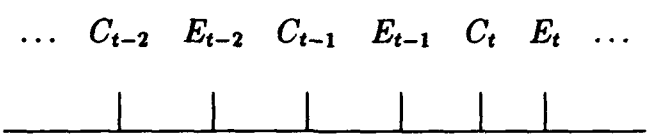
Let $P_1, \cdots, P_N$ be the N processes in the system. The processes coordinate their activities by use of a shared test-and-set variable $v = (b_{even}, r_{even}, b_{odd}, r_{odd}, s, p, w, partic)$ where each b component is $B = \log_2 N + 4$-valued, and the other components are $0, 1$ valued. Each process P_i has local variables $b^{(i)}, r^{(i)}, d^{(i)}$ and some flags. Variables local to P_i are denoted $x^{(i)}$.

The processes will use a geometric distribution to pick numbers between 1 and B. Namely, each $1 \leq i \leq B-1$ is drawn with probability $\frac{1}{2^i}$ and B is drawn with probabil-

ity $\frac{1}{2^{B-1}}$. We shall denote the act of drawing such a number by $b^{(i)} := randoml$. It was shown in [4] that for any $1 \leq m \leq N$ processes, say $P_1, \cdots, P_m$, if each P_i randomly, with a geometric distribution, draws a number $1 \leq b^{(i)} \leq B$, then with probability at least $2/3$ $b^{(j)} = \max b^{(i)}$ will hold for *exactly* one j.

During the computation, each process P_i is in one of four possible phases: *Trying phase*, in which it attempts to enter the critical section, *Critical phase*, in which it executes the critical section, *Exit phase*, in which it leaves the critical section, or *Remainder phase*, in which it does local computations. In this paper, we assume the same adversary scheduler that was postulated in [4]. Namely, at any given time the adversary scheduler can observe the *external behavior* of the processes (i.e., which of the four phases each process currently executes), and use this information (together with its information on the past behavior of the processes) to determine which process will be the next to access the shared variable. The adversary scheduler *cannot* observe the content of the shared variable nor the content of any local variable. More formally, let a *run* be a (finite or infinite) sequence $(i_1, x_1), \ldots, (i_k, x_k), \ldots$, where x_j indicates which phase process p_{i_j} started or whether it accessed the shared variable. A *scheduler* is a (probabilistic) function that on a finite run σ gives the name of the next process to access the shared variable. A run is *proper* if it satisfies the obvious consistency conditions.

The whole computation by $P_1, \cdots, P_N$ leading to entrance into the critical section is organized in intervals (the E_t, C_t are *logical*, not physical, times).



where $[C_t, E_t)$ is the t-th critical section, with C_t (and $s := 1$) marking the entry, E_t (and $s := 0$) marking the exit from this section.

In [4], the processes P_i trying to enter the next critical section $[C_t, E_t)$ drew, during the interval $[C_{t-1}, C_t)$, tickets $b^{(i)} := randoml$ and posted the result in v by $b := \max(b, b^{(i)})$. During $[E_{t-1}, C_t)$ the first process to arrive with a highest ticket $b^{(i)} = b$ enters the critical section. The crucial modification in the present algorithm is the following. In the t-th *drawing round* $[C_{t-2}, C_{t-1})$, participating processes draw numbers to determine the winning process P_i enabled to enter the t-th critical section at time C_t. By time C_{t-1}, all but at most B of the participants in the drawing round $[C_{t-2}, C_{t-1})$ will know that they lost in this drawing. In the t-th *notification round* $[C_{t-1}, C_t)$ each of the remaining participants finds out whether it won or lost, thus by time C_t there will be a P_i knowing that *it alone* won entrance to the critical section. The whole computation is arranged so that, for every t, the $t+1$-st drawing round overlaps with the t-th notification round, as indicated in Figure 1.

In other words, the interval $[C_{t-1}, C_t)$ serves both as the t-th notification round and as the $(t+1)$-th drawing round. To disambiguate this dual function of an interval $[C_{t-1}, C_t)$, the intervals are classified as even and odd. The parity component p of v will satisfy in the above interval $p = (t-1) \bmod 2$. Thus $p = 0$ signifies an even drawing round for entrance in the following even critical section at C_{t+1}, as well as an odd notification round to notify the winner in $[C_{t-2}, C_{t-1})$ - the just previous odd drawing round.

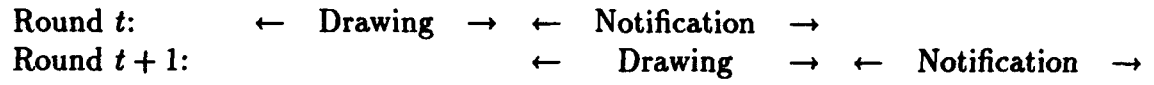


Figure 1: Organization of the computation in rounds

The key point is that since the drawing determining the winner P_i enabled to enter at C_t *ends* at time C_{t-1}, before the opening of the critical section at E_{t-1}, there will be no possibility for the scheduler to infer significant information about processes' local variables or about the r component of v while the drawing round $[C_{t-2}, C_{t-1})$ is in progress. It will also be seen that all participants losing in a drawing leave that drawing with their local lottery ticket = 0. Thus the scheduler cannot hoard processes with large ticket values and use them in later drawing rounds, as it could in the original version [4] of the algorithm.

2.2 Detailed Description of the Algorithm

When P_i is scheduled to test and set v, it knows by looking at local and global flags whether it should execute in some drawing round $[C_{t-1}, C_t)$ and, if yes, whether $t-1$ is even or odd. Assume P_i is in an even drawing round. It executes the protocol of Figure 2.

The test (1) in that protocol ensures that P_i will not draw more than once in $[C_{t-1}, C_t)$. If in the drawing it did not exceed the current value b_{even}, then it knows it lost and leaves with $d^{(i)} = b^{(i)} = 0$. Later it will not execute Notification, only Drawing.

Assume that $P_1, \ldots, P_k$ participated in $[C_{t-1}, C_t)$ and executed (2)-(3) (i.e., each of them at the time he drew his number was a local maxima). Then by C_t we have $b_{even} = d^{(1)} + \ldots + d^{(k)}$ (intuitively, $d^{(i)}$ is the contribution of P_i to b_{even}) and the unique winner is actually already determined. In particular, it follows that the number of processes executed (2)-(3) is small (in fact, it cannot be larger than B). All the other processes that participated in the drawing round already know they lost. This is very important because in the claimed size of v we cannot count, for example, the number of processes which participated in the lottery. The way we make sure that all the processes $P_1, \ldots, P_k$ will take part in the notification round is by letting each P_i to subtract its contribution $d^{(i)}$ from b_{even}. When all of them are notified we will have $b_{even} = 0$.

Notifying the winner and the losers amongst $P_1, \ldots, P_k$, is done in the even notification round $[C_t, C_{t+1})$. At time C_t we have $s = 0, p = 1$ (an odd drawing round) and $w = 0$ (winner not yet notified). The structure of the Notification round is as follows: the processes who lost are waiting until the winner is notified. Then, the losing processes are all notified and only then the winner is enabled to enter the critical section (if it is open). By looking at flags, every P_i knows whether for it $[C_t, C_{t+1})$ is a Notification round (or a Drawing round). It executes the protocol of Figure 3.

After P_i is enabled it must wait until the critical section will be open. It executes the protocol of Figure 4. The salient point in that protocol is that the round count number, r_{even}, is set to a *random* value. It is on this feature that the following proof of Lemma 1 rests.

Note that $partic = 1$ at C_{t+1} if and only if

```
    Even Drawing for P_i (p = 0):
(1) If [r^(i) ≠ r_even ∨ b_even = 0] ∧ [d^(i) = 0 ∧ w^(i) = 0] then   (* P_i has not drawn in [C_{t-1}, C_t)
                                                                        and is not participating in the
                                                                        current odd notification *)
    begin
        r^(i) := r_even
        b^(i) := randoml
        If b_even < b^(i) then                                        (* P_i is a local maxima *)
(2)         d^(i) := b^(i) - b_even
(3)         b_even := b^(i)
        else                                                          (* P_i knows it lost *)
            b^(i) := 0
    end
```

Figure 2: Even Drawing Protocol

```
    Even Notification for P_i (p = 1):
    If w = 0 ∧ b^(i) = b then                                         (* P_i is the winner *)
    begin
        w^(i) := 1                                                    (* P_i knows it won *)
        w := 1                                                        (* winner was notified *)
        b_even := b_even - d^(i)
        d^(i) := 0
    end
    If w = 1 ∧ w^(i) = 0 then                                         (* P_i knows it lost *)
    begin
        b_even := b_even - d^(i)
        b^(i) := 0                                                    (* clean up local variables *)
        d^(i) := 0
        r^(i) := nil
        P_i executes Odd Drawing
    end
    If w^(i) = 1 ∧ b_even = 0 then                                    (* all losers were notified *)
        P_i is enabled to enter the critical section
```

Figure 3: Even Notification Protocol

```
Enabled P_i Enters Even Critical Section:
If s = 0 then                          (* critical section is open *)
begin
        w := 0; w^(i) := 0
        b^(i) := 0                     (* clean up local variables *)
        d^(i) := 0
        r^(i) := nil
        r_even := random(0,1)          (* Assign 0 or 1 with equal probabilities *)
        p := 0                         (* start of even Drawing *)
        s := 1                         (* critical section closed *)
        If b_odd > 0 then              (* some process drew in [C_t, C_t+1) *)
                partic := 1            (* there was participation *)
        else
                partic := 0            (* no participation *)
end
```

Figure 4: Entrance to Even Critical Section

some P_j drew in $[C_t, C_{t+1})$. The above protocols should be augmented by the provision that a trying process P_i accessing v, upon finding $partic = 0$, sets $partic := 1, w := 1, w^{(i)} := 1$ and is enabled to enter the next critical section. In the beginning, v is initialized with $partic := 0$.

2.3 Correctness

It goes without saying, that the adversary scheduler can always discriminate against a chosen P_i by consistently scheduling it very rarely or scheduling it together with many other processes. However, given that P_i participates in a drawing round with $m-1$ other processes, the following lemma gives a lower bound on the probability of P_i to enter the critical section which depends only on m, the *actual* number of participants in the drawing round (and not on N). It is important to note that this probability is also independent of the past.

Before formulating the lemma, we need to argue that the probability space that we are dealing with is well defined. We say that a (proper) run $\sigma = (i_1, x_1), \ldots, (i_k, x_k)$ is of *length* t if exactly t of the x_j's indicating a start of a Critical phase, and one of them is x_k. Let V_{t-1} be the *view* of the system at time C_{t-1}, just before r_{even} was randomly set to 0 or 1 (we assume that $t-1$ is even). That is, V_{t-1} consists of the values of the shared variable and all the local variables in the system at this time. Given σ, a (proper) run of length t, and a view V_{t-1} *consistent* with it (e.g., the process that enters the critical section at time C_t should be the winner of the previous drawing round $[C_{t-2}, C_{t-1})$ as reflected by V_{t-1}), the winner of the drawing round $[C_{t-1}, C_t)$ depends only on the random choices of the processes which are in a Trying phase during this time, and the random value of r_{even}. It is important to note that the process that will enter the critical section at time

C_t is already determined by V_{t-1}. The random choices made during the time $[C_{t-1}, C_t)$ affect the values of the shared variable and the local variables, but will affect the external behavior of the processes only at time C_{t+1} (until that time they all stay in the Trying phase). Therefore, the run σ is independent of the random choices made during this time, and hence we get a well defined probability space. (In a sense, we give here an additional power to the adversary by allowing him a total view of the system just before the drawing round starts.) Now, we can formalize the lemma:

Lemma 1: Let σ be any (proper) run of length t, and V_{t-1} be any view consistent with σ. If in σ process P_i participates in the drawing round $[C_{t-1}, C_t)$ together with $m-1$ other processes and if V_{t-1} is the view of the system at time C_{t-1}, then with probability at least $1/3m$ the process P_i will enter the critical section at time C_{t+1}.

Proof: Assume $t-1$ is even. At time C_{t-1} the value of the component r_{even} was randomly set to 0 or 1.

Clearly, all processes which do not participate in the drawing round $[C_t, C_{t-1})$ have no influence on the drawing (they either access only the fields of the shared variable connected with the odd drawing rounds or do not access the shared variable at all). It is also guaranteed by the algorithm that every process P_j that first joins the drawing has $b^{(j)} = 0$.

As claimed above, even though we allow the scheduler full information on the past (by giving him V_{t-1}), and in particular we allow him to look at $r^{(j)}$ of all P_j's, the run σ is independent of the value of r_{even}, and in particular is independent of whether $r^{(i)} \neq r_{even}$ at the time that P_i first access v during the drawing round. Hence, as r_{even} chosen to be 0 or 1 uniformly, if $r^{(i)} \neq nil$ then with probability $1/2$ we have $r^{(i)} \neq r_{even}$ when P_i first accesses v in $[C_{t-1}, C_t)$. In such a case P_i draws a new $1 \leq b^{(i)} \leq B$ in this round. (If $r^{(i)} = nil$ then P_i draws a new number in this round with probability 1.)

Every other process P_j of the m participating processes draws its random $b^{(j)}$ at most once, during the current drawing round. Those P_j's which do not actually draw have $b^{(j)} = 0$ and do not affect b_{even} at all. By Rabin's lemma (quoted in the Introduction), the probability of having a unique winner is at least $2/3$ (no matter what is the number of other processes that draw new numbers). Given that there is a unique winner then, by symmetry argument, each process that draws a new number has the same probability to be the winner. (Note that there is no symmetry in case that the winner is not unique; the process that draws the maximal number first is the winner.) All together, with probability at least $1/2 \times 2/3 \times 1/m = 1/3m$, process P_i will draw the maximum value in this drawing round and will be alone in this. Hence $b^{(i)} = b_{even}$ at time C_t and P_i must be the one to enter the next critical section at C_{t+1}. □

The property that for any $m \leq N$ if P_i is participating in a drawing round together with any $m-1$ other processes, then with probability $1/\gamma m$ it will win entrance to the critical section, is called in [4] *γ-bounded waiting*. Thus we have

Theorem 2 : Mutual exclusion with bounded waiting can be achieved for a N processor system by use of an $O((\log_2 N)^2)$-valued shared test-and-set variable.

This is Theorem 4 of [4] except that now we require a $(\log_2 N)^2$-valued variable, instead of $\log_2 N$ values. (In terms of the number of

bits, both the old and new theorems use an $O(\log \log N)$-bit variable.)

3 Concluding Remarks

Another version of the mutual exclusion algorithm given in [4] is based on a random drawing suggested by Ben-Or. Specifically, P_i draws the value 2 with probability $1/N$ and the value 1 with probability $1 - 1/N$. It is readily seen that if P_i participates in a drawing together with $m - 1$, $1 \leq m < N$, other processes, then with probability at least $1/2eN$ the process P_i will be the sole winner. Using this lottery we get

Theorem 3: For an appropriate constant c ($c \leq 3^2 \cdot 2^6$), starvation-free mutual exclusion for N processes can be achieved by use of a c-valued test-and-set variable.

This is the same result as in [4]. As in [4], Theorems 2 and 3 should be contrasted with the lower bound results in [1]. It is shown there that for *deterministic* algorithms an N-valued variable is necessary for mutual exclusion with bounded waiting (in the sense of [1]), and an $N/2$-valued variable is necessary for starvation-free mutual exclusion.

In fact, the last algorithm can be further simplified: in the Notification round all that is needed is for the unique winner to find out that he won. The numbers held by the losing processes are at most 1 and therefore if P_i participates in a drawing round then still with probability at least $1/2eN$ the process P_i will be the sole winner. Also, in the first algorithm b can be duplicated (thus, increasing the number of bits in v by a constant factor) so that the losing processes will not have to wait until the winner is notified. This may be helpful in scenarios where some of the processes are much "slower" than the others.

An interesting open problem is to try to show a better upper bound than the one presented in Theorem 2, or to prove a lower bound. A step in this direction was achieved in [3] where it is proven that there is no better lottery with the unique winner property. Therefore, if one tries to improve the upper bound he cannot hope to do that by using only a different lottery, but instead he should find a somewhat different algorithm.

References

[1] Burns, J. E., M. J. Fischer, P. Jackson, N. A. Lynch, and G. L. Peterson, "Data Requirements for Implementation of N-Process Mutual Exclusion using a single shared variable", *JACM*, Vol. 29, 1982, pp. 183-205.

[2] Dijkstra, E., "Solution of a Problem in Concurrent Programming Control", *CACM*, 321, 1966.

[3] Kushilevitz E., Y. Mansour, and M. O. Rabin, in preparation.

[4] Rabin, M. O., "N-Process Mutual Exclusion with Bounded Waiting by $4 \log_2 N$-Valued Shared Variable", *JCSS*, Vol. 25 (1), 1982, pp. 66-75.

[5] Saias, I., and N. Lynch, "Proving Probabilistic Correctness Statements: The Case of Rabin's Algorithm for Mutual Exclusion", This Proceedings.

Errata: Knowledge in Shared Memory Systems
(PODC 1991)

Michael Merritt Gadi Taubenfeld

The errata: In subsection 3.2, after the definition of knowledge, appears the following claim: "Notice that when β is a stable predicate then also $K_{\mathcal{G}}\beta$ is a stable predicate." This claim is incorrect, and should be replaced with: "Notice that when β is a stable predicate then $K_{\mathcal{G}}\beta$ **is not neccessarily** a stable predicate."

The proofs of Theorem 1 and Theorem 2 relied on this incorrect claim, and are correct only for β for which $K_{\mathcal{G}}\beta$ is stable. Following are new proofs of Theorem 1 and Theorem 2 that hold in the general case, in which $K_{\mathcal{G}}\beta$ may not be stable.. The authors regret the error, and thank Yoram Moses and Lenore Zuck for pointing it out.

Recall that it is assumed in the two theorems that β_1 and β_2 are disjoint stable predicates.

Theorem 1 *In any asynchronous read-write protocol, for any run x, if $\Delta\{\beta_1,\beta_2\}$ at x then for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$, $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x.*

In order to prove the theorem we first prove two lemmas. Without loss of generality, we consider in this proof only deterministic processes. That is, if $\langle x; e_p\rangle$ and $\langle x; e'_p\rangle$ are runs then $e_p = e'_p$. When p is enabled at x, we denote by $\circ_p x$ the unique extension of x by a single event of p. Finally, by $\bar{\mathcal{G}}$ we denote the set of all processes not in $\mathcal{G}$.

Lemma 1 *Let x and y be finite runs and $\mathcal{G}$ be a set of processes. If $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta_1$ at x, $x[\mathcal{G}]y$, and $value(r,x) = value(r,y)$ for every shared register r, then $\neg\Diamond\beta_2$ at y.*

Proof: Assume $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta_1$ at x, $x[\mathcal{G}]y$, and $value(r,x) = value(r,y)$ for every shared register r. From $RW1$–$RW2$, x is a prefix of a $\mathcal{G}$-fair run z where $x[\bar{\mathcal{G}}]z$. Since $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta_1$ at x there is a finite run $x \le x' \le z$ such that $x[\bar{\mathcal{G}}]x'$ and $K_{\mathcal{G}}\beta_1$ at x'. From $RW1$–$RW3$, $w = \langle y; (x'-x)\rangle$ is also a run. Since $K_{\mathcal{G}}\beta_1$ at x' and $x'[\mathcal{G}]w$, it has to be that β_1 at w. Since β_1 is stable and disjoint from β_2, it is the case that $\neg\Diamond\beta_2$ at y. ∎

We use the notation $(\Diamond K)_{n'}\beta$ at x as an abbreviation for $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n'$. Notice that it follows from Lemma 1 that for any finite runs x and y and set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n'$, if $(\Diamond K)_{n'}\beta_1$ at x, $x[\mathcal{G}]y$, and $value(r,x) = value(r,y)$ for every shared register r, then $\neg\Diamond\beta_2$ at y.

Lemma 2 *For any finite run x and any process p, if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n-1$, $\Delta\{\beta_1,\beta_2\}$ at x, and p is enabled at x then there exists $y \ge x$ such that $\neg x[p]y$ and $\Delta\{\beta_1,\beta_2\}$ at y.*

Proof: Assume to the contrary that for some run x and process p, $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n-1$, $\Delta\{\beta_1,\beta_2\}$ at x, p is enabled at x and there does not exist $y \ge x$ such that $\neg x[p]y$ and $\Delta\{\beta_1,\beta_2\}$ at y. Observe that it follows from the assumption that for any extension m of x where $\neg(\beta_1 \vee \beta_2)$ at m and p is enabled at x, either $(\Diamond K)_{n-1}\beta_1$ at $\circ_p m$ or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p m$. Assume w.l.o.g. that $(\Diamond K)_{n-1}\beta_1$ at $\circ_p x$.

Since $\Delta\{\beta_1,\beta_2\}$ at x there exists a finite extension z of x ($z \ne x$) such that $\Diamond\beta_2$ at z and for any $x \le y < z$ it is the case that $\Delta\{\beta_1,\beta_2\}$ at y. It is important to notice that $(\Diamond K)_{n-1}\beta_2$ at z. Furthermore, either β_2 at z or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p z$ (if $\circ_p z$ exists).

Let z' be the longest prefix of z such that $x[p]z'$. We notice that either β_2 at z' (when $z' = z$) or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p z'$, and in either case, $\Diamond\beta_2$ at $\circ_p z'$.

Consider the extensions of x which are also prefixes of z'. Since $(\Diamond K)_{n-1}\beta_1$ at $\circ_p x$ by the observations made so far, there must exist extensions y and y' (of x) where y is a one event extension of y', such that $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$, either β_2 at y (when $y = z$) or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$, and in either case, $\Diamond\beta_2$ at $\circ_p y$. Let $y = \langle y'; e_{p'}\rangle$ for some event $e_{p'}$ where $p' \ne p$.

First we show that $e_{p'}$ is a write event. Assume that $e_{p'}$ is not a write event. By $RW1$–$RW3$, $(\circ_p y' - y') = (\circ_p y - y)$ and hence $\circ_p y'[N-\{p'\}]\circ_p y$. Also, the values of all shared registers are the same in $\circ_p y$ and $\circ_p y'$, and as already mentioned $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$. By Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y'$, a contradiction. Therefore, $e_{p'}$ must be a write event.

We notice that, by $RW1$ $w = \langle \circ_p y'; e_{p'}\rangle$ is a run. Also, since $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$, it must be the case that for $\mathcal{G} = N - \{p'\}$, $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta_1$ at w.

Next we show that $(\circ_p y' - y')$ is a write event. Assume $(\circ_p y' - y')$ is not a write event. Then, $w[N-\{p\}]y$, and $w[N-\{p\}]\circ_p y$. Also, the values of all shared registers are the same in w, y and $\circ_p y$. Because $w \ge \circ_p y'$, $\Diamond\beta_1$ at w. Now, recall that either β_2 at y or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$. It follows that, either $(\Diamond K)_{n-1}\beta_2$ at y or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$. By applying Lemma 1, in both cases $\neg\Diamond\beta_1$ at w, a contradiction. Therefore, for two registers r_1 and r_2, and values v_1 and v_2, $(\circ_p y' - y') = write_p(r_1,v_1)$, and $(y - y') = write_{p'}(r_2,v_2)$.

Assume $r_1 \ne r_2$. Since the two write events are independent, the values of all shared registers are the same in w and $\circ_p y$. Also, $w[N]\circ_p y$ and hence $w[\mathcal{G}]\circ_p y$ for $\mathcal{G} = N - \{p'\}$. As pointed out, $\Diamond_{\mathcal{G}}K_{\mathcal{G}}\beta_1$ at w, and hence by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

Assume $r_1 = r_2$. Clearly, $value(r_1, \circ_p y') = value(r_1, \circ_p y)$. Hence, the values of all shared registers are the same in $\circ_p y'$ and $\circ_p y$. Also, $\circ_p y'[N-\{p'\}]\circ_p y$ and as assumed $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$. As before by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction. ∎

Proof of Theorem 1: Assume to the contrary that for some run x where $\Delta\{\beta_1,\beta_2\}$ at x, it is the case that $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee$

$K_{\mathcal{G}}\beta_2$) at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$. Using Lemma 2 we can construct inductively starting from the run x an n-fair run such that $\Delta\{\beta_1, \beta_2\}$ holds at all the finite prefixes of that run, and $\neg\Diamond_N(K_N\beta_1 \vee K_N\beta_2)$ at x. Thus, $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$, a contradiction. ∎

Theorem 2 *In any asynchronous read-modify-write protocol, for any run x where $\Delta\{\beta_1, \beta_2\}$ at x, if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$, then the protocol uses at least one non-binary shared register.*

In order to prove the theorem we first prove two lemmas. In proving these lemmas we assume that all shared registers are binary registers. As in Theorem 1, without loss of generality, we consider in this proof only deterministic processes, and denote by $\circ_p x$ the unique extension of x by a single event on p. We point out that Lemma 1, and the observation following it, although proved for read-write protocols, hold also for read-modify-write protocols with essentially the same proof.

Lemma 3 *Let x be a run, p and p' be two processes which are enabled at x, and $\mathcal{G} = N - \{p, p'\}$. If $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$ then $\neg\Diamond\beta_2$ at $\circ_{p'} x$.*

Proof: Assume $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$ and let $(\circ_p x - x) = rmw_p(r_1, v_1', v_1)$, and $(\circ_{p'} x - x) = rmw_{p'}(r_2, v_2', v_2)$, for registers r_1 and r_2, and values $v_1', v_1, v_2', v_2 \in \{0,1\}$. Let $y = \circ_{p'} \circ_p x$ and $y' = \circ_p \circ_{p'} x$. Note that, since $y \geq \circ_p x$ and $y[\mathcal{G}] \circ_p x$, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y.

Assume $r_1 \neq r_2$. Since the two events are independent, the values of all shared registers are the same in y and y', and $y[N]y'$. By the note above, also $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y. Thus, by Lemma 1, $\neg\Diamond\beta_2$ at y', and hence $\neg\Diamond\beta_2$ at $\circ_{p'} x$.

Assume $r_1 = r_2$. Then $v_1' = v_2'$. There are three possible cases. (1) $v_1 = v_2$. Since, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$, $\circ_p x[\mathcal{G}]\circ_{p'} x$, and $value(r, \circ_p x) = value(r, \circ_{p'} x)$ for every shared register r, by Lemma 1 $\neg\Diamond\beta_2$ at $\circ_{p'} x$. (2) $v_1' = v_1$. By $RMW1$, $y = \langle \circ_p x; rmw_{p'}(r_2, v_2', v_2)\rangle$ is a run. Again by the note above, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y, $y[\mathcal{G}]\circ_{p'} x$, and $value(r, y) = value(r, \circ_{p'} x)$ for any shared register r, by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_{p'} x$. (3) $v_2' = v_2$. The values of all shared registers are the same in $\circ_p x$ and y', $\circ_p x[\mathcal{G}]y'$, and also $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$. Thus, by Lemma 1, $\neg\Diamond\beta_2$ at y', and hence $\neg\Diamond\beta_2$ at $\circ_{p'} x$. ∎

In the proof of the next lemma the notation $(\Diamond K)_{n'}\beta$ from the previous section is used.

Lemma 4 *For any run x and any two processes p and p' which are enabled at x,*
if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$ and $\Delta\{\beta_1, \beta_2\}$ at x then there exists $y \geq x$ such that $\neg x[\{p, p'\}]y$ and $\Delta\{\beta_1, \beta_2\}$ at y.

Proof: Assume to the contrary that for some run x and two processes p and p' which are enabled at x, $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$, $\Delta\{\beta_1, \beta_2\}$ at x, and there does not exist $y \geq x$ such that $\neg x[\{p, p'\}]y$ and $\Delta\{\beta_1, \beta_2\}$ at y. Observe that it follows from the assumption that for any extension m of x where $\neg(\beta_1 \vee \beta_2)$ at m and both p and p' are enabled at m, that either $(\Diamond K)_{n-2}\beta_1$ at $\circ_p m$ or $(\Diamond K)_{n-2}\beta_2$ at $\circ_p m$, and either $(\Diamond K)_{n-2}\beta_1$ at $\circ_{p'} m$ or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} m$. Since $\neg(\beta_1 \vee \beta_2)$ at x, assume w.l.o.g. that $(\Diamond K)_{n-2}\beta_1$ at $\circ_p x$. By Lemma 3, also $(\Diamond K)_{n-2}\beta_1$ at $\circ_{p'} x$.

Since $\Delta\{\beta_1, \beta_2\}$ at x, there exists an extension z of x ($z \neq x$) such that $\Diamond\beta_2$ at z and for any $x \leq y < z$. it is the case that $\Delta\{\beta_1, \beta_2\}$ at y. It is important to notice that $(\Diamond K)_{n-2}\beta_2$ at z. Furthermore, either β_2 at z or both $(\Diamond K)_{n-2}\beta_2$ at $\circ_p z$ (if $\circ_p z$ exists) and $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} z$ (if $\circ_{p'} z$ exists).

Let z' be the longest prefix of z such that $x[\{p, p'\}]z'$. We notice that, by RMW2, for any $x \leq y \leq z'$ both p and p' are enabled at y. Since $z' \leq z$, it follows from the first assumption and Lemma 3 that either β_2 at z' (when $z' = z$) or $(\Diamond K)_{n-2}\beta_2$ at $\circ_p z'$ and $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} z'$. In any case, $\Diamond\beta_2$ at both $\circ_p z'$ and $\circ_{p'} z'$.

Consider the extensions of x which are also prefixes of z'. Since $(\Diamond K)_{n-2}\beta_1$ at $\circ_p x$ there must exist extensions y and y' (of x) where y is a one event extension of y', such that $(\Diamond K)_{n-2}\beta_1$ at both $\circ_p y'$ and $\circ_{p'} y'$, and either β_2 at y (when $y = z$) or $(\Diamond K)_{n-2}\beta_2$ at both $\circ_p y$ and $\circ_{p'} y$, and in either case, $\Diamond\beta_2$ at both $\circ_p y$ and $\circ_{p'} y$. Let $y = \langle y'; e_{p''}\rangle$ for some event $e_{p''}$ where $p'' \notin \{p, p'\}$.

For some registers r_1 and r_2, and values $v_1', v_1, v_2', v_2 \in \{0,1\}$, $(\circ_p y' - y') = rmw_p(r_1, v_1', v_1)$, and $(y - y') = rmw_{p''}(r_2, v_2', v_2)$.

Assume $r_1 \neq r_2$. By $RMW1$, $w = \langle y'; rmw_p(r_1, v_1', v_1); rmw_{p''}(r_2, v_2', v_2)\rangle$ is a run. The values of all shared registers are the same in w and $\circ_p y$, and $w[N]\circ_p y$. Let $\mathcal{G} = \{N - p''\}$, because $w > \circ_p y'$ and $w[\mathcal{G}]\circ_p y'$, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at w. By Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

Assume $r_1 = r_2$. Then $v_1' = v_2'$. There are three possible cases. (1) $v_2' = v_2$. Since $(\Diamond K)_{n-2}\beta_1$ at $\circ_p y'$, $\circ_p y'[N - \{p''\}]\circ_p y$, and the values of all shared registers are the same in $\circ_p y'$ and $\circ_p y$, by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

(2) $v_1' = v_1$. Let $\mathcal{H} = N - \{p, p'\}$. Recall that either β_2 at y or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} y$. It follows that, either $(\Diamond K)_{n-2}\beta_2$ at y or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} y$, and in either case $\Diamond_{\mathcal{H}} K_{\mathcal{H}}\beta_2$ at $\circ_{p'} y$. Let $s = \circ_{p'} \circ_{p''} \circ_p y'$. Because $s > \circ_p y'$, $\Diamond\beta_1$ at s. Since $\circ_{p'} y[\mathcal{H}]s$, and the values of all shared registers are the same in $\circ_{p'} y$ and s, by Lemma 1, $\neg\Diamond\beta_1$ at s, a contradiction.

(3) $v_1 = v_2$. Recall that either β_2 at y or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} y$. It follows that, either $(\Diamond K)_{n-2}\beta_2$ at y or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} y$. Assume $(\Diamond K)_{n-2}\beta_2$ at y. Since $y[N - \{p, p''\}] \circ_p y'$, and the values of all shared registers are the same in y and $\circ_p y'$, by Lemma 1, $\neg\Diamond\beta_1$ at $\circ_p y'$, a contradiction. Assume $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} y$. Let $\ell = \circ_{p'} \circ_p y'$. Because $\ell > \circ_p y'$, $\Diamond\beta_1$ at ℓ. Since $\circ_{p'} y[N - \{p, p''\}]\ell$, and the values of all shared registers are the same in $\circ_{p'} y$ and ℓ, by Lemma 1, $\neg\Diamond\beta_1$ at ℓ, a contradiction. ∎

Proof of Theorem 2: Assume to the contrary that for some run x where $\Delta\{\beta_1, \beta_2\}$ at x, it is the case that $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-2$. Using Lemma 4 we can construct inductively starting from the run x an $(n-1)$-fair run such that $\Delta\{\beta_1, \beta_2\}$ holds at all the finite prefixes of that run. Thus, $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-2$, a contradiction. ∎

Author Index